A MODEL OF THE ADA AVIONICS REAL-TIME SYSTEM:
An Example of the Benefits of the Hardware/Software Codesign Approach in Development of Real-Time Systems

Prepared by:

B. E. Clark
F. G. Gray
J. T. Morrison
T. S. White

TRW Military Electronics and Avionics Division
Dayton Engineering Laboratory
Beavercreek, Ohio

March 1992

Prepared by:

Center for Digital Systems Research
Research Triangle Institute
Research Triangle Park, North Carolina 27709

and

Virginia Polytechnic Institute
Blacksburg, Virginia

AVIONICS DIRECTORATE
WRIGHT LABORATORY
AIR FORCE SYSTEMS COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OHIO 45433-6543

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation; or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.

This technical report has been reviewed and is approved for publication.

Joseph S. Wilgus

JOSEPH S. WILGUS, Project Engineer
Advanced Integration Group
System Avionics Division
Avionics Directorate

David A. Zann

DAVID A. ZANN, Chief
System Integration Branch
System Avionics Division
Avionics Directorate

Charles H. Krueger

CHARLES H. KRUEGER, Chief
System Avionics Division
Avionics Directorate

If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/AAAS, WPAFB, OH 45433-6543 to help us maintain a current mailing list.

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
March 1992

3. REPORT TYPE AND DATES COVERED
Final Report

4. TITLE AND SUBTITLE
A Model of the Ada Avionics Real-Time System: An Example of the Benefits of the Hardware/Software Codesign Approach in Development of Real-Time Systems

5. FUNDING NUMBERS
F33615-87-D-1452
PE 62204F
PR 3062
TA 01
WU 11

6. AUTHOR(S)
B.E. Clark, F.G. Gray, J.T. Morrison, and T.S. White

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Research Triangle Institute
Center for Digital Systems Research
P.O. Box 12194
Research Triangle Park, NC 27709

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Joseph S. Wilgus (513) 255-4709
Avionics Directorate (WL/AAAS)
Wright Laboratory
Wright-Patterson AFB, OH 45433-6543

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
WL-TR-92-1022

11. SUPPLEMENTARY NOTES
Prepared under Subcontract No. FF9327VB0S
Prime Contract (USAF) F33615-87-D-1452

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
The Ada Avionics Real-Time System (AARTS) Operating System (AOS) is the OS and management system under development for the PAVE PILLAR architecture. The AOS, in its current version, was modeled in the Architectural Design and Assessment Systems (ADAS) along with the hardware and applications being exercised in the Avionics Directorate's Integrated test bed. The report describes the model, the results of simulation executions, and methods for expansion of the model to architectures larger than that of the integrated test bed.

14. SUBJECT TERMS
ADAS, Avionics, Modeling, PAVE PILLAR, Simulation, VAMP, VHSIC Avionics Multiprocessor

15. NUMBER OF PAGES
184

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18

Contents

List of Figures

List of Tables

Symbols and Abbreviations

AARTS	Ada Avionics Real Time System
ABI	Avionics Bus Interface
ADAS	Architecture Design and Assessment System
AS	Address States
BIT	Built-in Test
BIU	Bus Interface Unit
BTBIM	Block Transfer Bus Interface Modules
BTB	Block Transfer Bus
CCB	Communication Control Block
CPU	Central Processing Unit
DA	Destination Address
DDA	Data Destination Address
DGS	Display Generation System
DMA	Direct Memory Access
ED	Error Detecting (PI-Bus), End Delimiter (HSDB)
ESA	Execution Start Address
FC	Frame Control
FCS	Frame Check Sequence
FIFO	First In First Out
HCCB	HSDB Communication Control Block
HSDB	High Speed Data Bus
I/O	Input/Output
IMFKey	Integrated Multifunction Display Key
IRS	Interface Requirements Specification
ISTC	Initialization Sequence Try Count
LPU	Loadable Program Unit
MAB	Mission Avionics Bus
MABIM	Mission Avionics Bus Interface Module
mips	Million Instructions per Second
MMKey	Mission Mode Key
SA	Source Address
SD	Start Delimiter
SMM	System Mass Memory
SUROM	Startup ROM
TLCSC	Top-Level Computer Software Components
TMT	Transmission Monitor Timer
TST	Transmission Streaming Timer
VAMP	VHSIC Avionics Multiprocessor
VHSIC	Very High Speed Integrated Circuit
VPI	Virginia Polytechnic Institute
WC	Word Count
WEC	Westinghouse Electric Company

1. Introduction

This report describes the model developed by the Research Triangle Institute (RTI) and Virginia Polytechnic Institute (VPI) under TRW subcontract Number FF9327VBOS. The objective was to develop and demonstrate an executable model of configuration and reconfiguration of the Ada Avionics Real Time System (AARTS) running on the Wright Laboratories VHSIC Avionics Multiprocessor (VAMP) demonstration hardware. The model was developed, using the Architecture Design and Assessment System (ADAS) delivered and executed using GIPSIM simulation. This report describes the model in detail and provides examples that show the usefulness of developing an executable model in parallel with system design. The modeling effort and model execution serve to validate (or invalidate) design decisions as they are made.

The model was constructed from information contained in numerous AARTS developmental documents and internal documents and reference furnished by the TRW project manager. This was reinforced by regular technical exchange discussions between the RTI and TRW project managers. In addition to two formal demonstrations of the model, a thorough review was conducted with the principle members of the AARTS development team in January 1991. This review and the attendance of the AARTS development team at the demonstrations served to verify, at each point, that the ADAS model is a true representation of the design as it exists (or is visualized) at that point.

This report is intended for users who want to employ the model for continuing analysis of AARTS development and expansion and, possibly, as a point of departure for follow-on developments. It is assumed that the reader is familiar with the basic AARTS architecture and functioning, as well as that of the VAMPs. The report focuses on the model – how functions are simulated, how resource utilization was estimated, and how the execution is controlled. These subjects are addressed in considerable detail to provide a reader, who is familiar with the ADAS tool set, with sufficient understanding of the model so that he can make the necessary changes to analyze the impact of changes in AARTS design, hardware capability, or function resource requirements. It should also provide a background for expansion of the model to a more extensive set of applications, a larger or more complex hardware architecture, or both.

The final three sections provide examples of model outputs and analyses, an approach for expanding the model, and the types of errors that can be detected early in the design with an ADAS modeling effort.

1.1. Background

Under contract from the U.S. Air Force, TRW is developing the AARTS operating system for Wright Laboratory. AARTS is the implementation of the PAVE PILLAR Operating System and system management concept. The AARTS is targeted for the laboratory VHSIC Avionic Modular Processors (VAMP) being developed by West-

inghouse Electric Company (WEC). The ADAS model developed in this effort is to be calibrated with AARTS Demonstration 3 and then, in a second phase, expanded to represent the entire PAVE PILLAR mission application architecture. GIPSIM simulation of the expanded model will provide an assessment of specific hardware and software partitioning needs to meet the PAVE PILLAR specification. Of particular interest is the Block Transfer Bus utilization and the total time it takes to complete various processes associated with system startup and reconfiguration following module failure.

The distributed architecture of PAVE PILLAR will provide for maximum utilization of common hardware and software programs, as well as providing maximum reliability, maintainability, and support for both air-to-air and air-to-ground missions. During development this model has served to highlight issues or deficiencies in early design decisions by addressing, in a system context, hardware/software interfaces. Once calibrated and validated against an implemented AARTS System it will provide the basis for simulation and analysis of the PAVE PILLAR architecture with developmental avionics processes integrated into an expanded suite of VAMP or VAMP-like clusters.

1.2. ADAS Model Of AARTS: An Overview

The Ada Avionics Real Time System (AARTS) is the evolving implementation of the PAVE PILLAR operating system concept. The AARTS is divided into three top-level computer software components (TLCSCs) called executives. The kernel ex-

ecutive TLCSC manages the resources of a single VHSIC module. It is the operating system for the module. The Distributed Executive TLCSC provides the services for communication between modules. Versions of the distributed executive for CPU, high-speed databus interface, M1553B bus interface and mass memory modules differ. The difference is normally the presence of a component associated with a specific interface (i.e., bus) or, in the case of CPU's, the functionality needed to establish message connections. The third major component of AARTS is the System Executive TLCSC. This component contains the software that manages the system. This component can function as a cluster supervisor, managing a cluster of modules; as the system supervisor; managing the system; or as a hot backup for the system supervisor. The targeted VHSIC Avionic Modular Processor (VAMP) consists of five VHSIC modules, each containing a 16-bit V1750A processor with 128 or 256K of memory. The memory in each module has been divided into numbered address states (AS)(i.e., AS0, AS1, etc.). The lowest address State, AS0, is reserved for the basic operating system. This consists of the kernel executive, the distributed executive and the kernel unit of the system executive. This is referred to as the AS0 Software throughout this report. (Most of the lower level software components of the AS0 software contain an I/O interface unit that is resident in any address State that can call for the TLCSC services).

Address states one and higher can contain any loadable program unit (LPU). The system executive (less the kernel) is loaded in AS1 by cluster supervisors upon winning

arbitration. Assumption of the role of System Supervisor or hot backup only requires enabling additional units. This Supervisory Software is referred to as the AS1 software through the remainder of this report.

The AARTS development program includes several demonstrations. These demonstrations progress from operating a single module through a several cluster effort with dynamic LPU loading. Demonstration 3 was to be conducted on two VHSIC Avionic Modular Processor (VAMP) clusters. The clusters are currently connected to a simulated System Mass Memory and to one another via high-speed fiber optic data busses. Each VAMP contains five processor modules that communicate with one another via a PI-bus. The five modules consist of two CPU modules, two high-speed databus interface modules and a M1553B bus interface module. Demonstration 3 was to start the system, execute a guidance and navigation scenario consisting of five LPUs, and demonstrate recovery from failure.

1.3. Acknowledgements

The development of this model was a team effort. RTI and its subcontractor, Virginia Polytechnic Institute and State University (VPI) produced the model. The TRW Dayton Engineering Laboratory managed the RTI contract and provided data descriptions of AARTS and Demonstration 3 without which the model could not have been developed. Four individuals deserve particular recognition. Professor F. Gail Gray, of the Department of Electrical Engineering at VPI, served as consultant to the

RTI principal investigator on methods for system abstraction and reviewed and critiqued the model at strategic points in the development. Dr. Tennis S. White, then at VPI, currently with IBM Glendale Research Laboratory, produced the actual graphs. Dr. White devised the control schemes to be described later in this report. Mr. J.L. Stautberg, of TRW Dayton Engineering Laboratory, served as project monitor and liaison between the modeling team and the AARTS development team. Finally, Mr. Joseph Wilgus of Wright Laboratory provided oversight and coordination of the entire effort, a task of much more significance and value than this simple statement can convey.

2. The ADAS Product

ADAS is a set of computer-aided design tools for the synthesis and analysis of software algorithms and their hardware implementations at the architectural level. ADAS models hardware and software using directed graphs in which nodes represent individual software operations or hardware functional elements and arcs represent data and control flow paths. Color-coded connection points called ports indicate the direction of flows along arcs. Nodes and arcs are typed and have a number of attributes associated with them. Nodes can be expanded into subgraphs that represent the refinement of a software operation or hardware component into a set of lower level operations or components and their interconnections. Nodes with subgraphs are called *internal nodes*; nodes without subgraphs are called *leaf nodes*.

Simulation is controlled on a graph or graph hierarchy by the movement of units called *tokens* around the graphs. During simulation, nodes produce and consume tokens and arcs act as FIFO queues of tokens.

Using this ADAS model, the user can test alternative algorithm and architecture strategies measuring performances, latency, timing, resource utilization, etc.

2.1. How the ADAS Tools Interact

Data flows between the ADAS tools and the data base files which are illustrated in Figure A-1. In this diagram, circles represent individual ADAS tools; directed lines

represent data flows between the tools and individual data base files, the contents of which are described between paired horizontal lines; and boxes represent analysis processes outputs. ADAS system.

The numbers on program circles in Figure A-1 represent the order in which the ADAS tools would typically be executed during a single design cycle. The design process typically consists of a number of iterations of the cycle. A design is analyzed and its execution simulated, and the results are used to modify and refine the design. This analysis/refinement process is repeated until the design's performance meets specification. Each circle in the diagram illustrates one or more phases of the design process:

1. EDIGRAF — The directed graph editor creates the initial template and graph data base files for the hardware and software design graphs; it is also used to make modifications to the templates, graph structure, and node and arc attributes throughout the iterative design process.
2. CONCH — The design graph consistency checker verifies that graph data flows are consistent (e.g., that component types match) and optionally checks graph attribute values against template values.
3. GIPSIM — The directed graph simulator performs initial verification of software graphs; it verifies that nodes are firing in the correct order, that token produce and consume values are correct, and that firing frequencies are approximately correct.
4. XPETRI — The performance analysis program generates petri net models of software directed graphs for detailed analysis of design performance. If the performance is not satisfactory, the software can be modified with EDIGRAF, and the design cycle repeated.
5. ASH — The task allocation tool assigns hardware graph components to software graph operations.

6-7. CSIM/ADASIM	A design graph functional simulator generation program constructs a program, CSIM or ADASIM, to simulate execution of the design from functional descriptions of the individual design graph nodes in the C or Ada programming language.
8. ISPGEN/HELIXGEN	Finally, a hardware design language generator constructs a program to simulate execution of the hardware design from functional descriptions of the individual hardware design graph nodes in the ISPS or HELIX language.

The individual files that form the ADAS common data base are shared by the tools that comprise the ADAS system and form the basis of the tools' integration into a coherent system for software/hardware codesign. The data base includes template data bases which contain representations of the basic building blocks that are used to construct graph data bases. The latter contain the data that define the positions and interconnections of nodes and arcs in software and hardware design graphs.

2.2. Graph, Node, Arc and Port Attributes

The behavior of the checking, mapping, and simulation tools is controlled by the topology (connectivity) of the graphs and by attributes assigned to the graphs, the nodes and their ports, and the arcs. The topology and attributes are assigned and/or modified using the (EDIGRAF) graphics editor. Initially, attributes are inherited from the default values contained in the template for the element. In this paragraph the attributes associated with the graph objects are reviewed to provide background for the discussion of the models in the remainder of the report.

For this discussion, attributes are divided into two major classes, those assigned by

the ADAS tools and not modifiable by the modeler and those assigned and modifiable by the modeler. The latter category has been divided into six subcategories and will be discussed first.

The first subcategory, display attributes, serve primarily to help make the graphs readable and understandable. These attributes are listed in Table 2.1.

Table 2.1. Display Attributes

Graph	Node	Arc
graph_name	node_name*	arc_name*
	node_color	arc_color
	node_height	first_joint
	node_width	....
	node orientation	fifth_joint
*Used by checking and simulation tools to identify output		

Each graph, node, and arc has a name attribute. The graph name is optional text and shows up as a banner at the top of the graph on the monitor. Any legal text entry can be chosen for a graph name. Legal ADAS text is defined on pages 3-30 of the ADAS User Manual [4]. The nodes and arcs are also named. These names must be unique (the default name is the template name followed by a unique number) since they are used extensively to identify output statistics from the ADAS tools. Node and arc names can be any unique legal ADAS label. Legal ADAS labels are defined on pages 3-30 of the ADAS User Manual [4]. A color can be selected for each node and arc from the ADAS 16 color palette. These colors are used to enhance understanding of

the graphs. The height and width of a node in grid units can be selected to improve appearance. ADAS nodes are constructed with all input ports on one side and all output ports on the opposite side. Node orientation representing the direction of flow through the node can be selected as down (default), up, right, or left. An arc can be drawn with a maximum of five joints (direction changes between the source outport and sink inport). The coordinates of these joints, if any, are stored as arc attributes.

The second subcategory, connectivity attributes, are used primarily to support conversion of a hierarchy of graphs into a single executable model. This model, consisting of all of the connected leaf nodes, is referred to as the flattened graph or flattened model. These attributes, shown in Table 2.2, are also used extensively by the checking and validation routines.

Table 2.2. Connectivity Attributes

Node	Node Port	Arc
node_class	inport_id	token_data_type
subgraf_file_name	outport_id	arc_template
graph_port_number	in_token_data_type	
node_template	out_token_data_type	

The attribute *node_class* indicates how the node functions in the model. It may have value of leaf, a node which maps to a specific hardware model; internal, a node which has a subgraph to be expanded within the executable model; or inport/outport, a node that represents an inport/outport on the parent level graph node. If a node is of node class internal it must have the file name for the subgraph in the attribute

subgraf_file_name. Port attributes are actually a part of the node data. A set of attributes for each input and output port is contained in each node database. Since these port attributes carry information critical to simulation and since there can be many ports on a given node, the port attributes have been placed in a separate column in the tables presented in this section. The first port attribute of interest is the *in(out)port_id*. Inports and outports of a node are uniquely identified by a number from zero to the number of in (out) ports minus one, clockwise for inports and counterclockwise for outports (left to right when orientation is down). On a subgraph of a node, there must be one graph port node for each inport and outport on the parent node. The *graph_port_number* is the number of the associated port on the parent node. Each in (out) port has an associated *in(out)_token_data_type*. This *data_type* may be any legal ADAS identifier. Each arc has a *token_data_type attribute*. This attribute must match the *data_type* attribute of both the source and sink ports before a connection can be made. Finally, the checking tools will warn one if the node or arch template identified in the attribute *node_template* or *arc_template* does not match the node or arc. This usually occurs when attributes of a node, port, or arc have been edited subsequent to creation and a new template has not been referenced and/or developed.

The third subcategory is those attributes of the software model employed by the simulation tools, particularly GIPSIM, which is the tool most applicable at this stage to the AARTS Demonstration Model. These attributes are listed in Table 2.3.

Table 2.3. Simulation Attributes

Graph	Node	Node Port	Arc
time_unit	hardware_module	firing_threshold	queue_size
conversion_factor	execution_order	token_produce_rate	
	firing_delay	token_consume_rate	
	trace_flag	initial_token_count	
	node_user_text		

Two attributes of the graph relate to the time steps used in a simulation. The attribute *time_unit* is a label documenting the units in which firing delays are calculated for the nodes. The attribute *conversion_factor* is a floating point number that relates the time unit for a subgraph to the time unit of the parent graph. The node attribute *hardware_module* for a software node contains the name of the node in the hardware model onto which the software node is mapped. This is a many to one mapping, i.e., any number software nodes may be mapped to a single hardware node (resources) but each software node may be mapped to only one hardware node. The software node attribute *execution_order* is an integer establishing queuing priority for solving hardware module contention during simulation. The node attribute *firing_delay* is the number of time units the node waits once it has received its resource until the resource is released. Alternatively, it is the time the node waits before firing (producing its output tokens) after it is primed. The attribute *trace_flag* is a flag that when set generates a simulation output listing the schedule of all primings and firings of a node. The attribute *node_user_text* may contain any text entered by the user. This attribute may be accessed by CSIM or AdaSIM. If its value is set to "any," all simulations treat

it as an OR-node (the node is enabled whenever <u>any</u> rather than <u>all</u> of its inports has reached its firing threshold). The node input port attributes *firing_threshold* and *token_consume_rate*, along with the output port attribute *token_produce_rate*, provide the data necessary to control and synchronize execution of the GIPSIM simulation. A node is enabled when each (any in the case of an OR-node) of its input arcs contains a number of tokens in its queue greater than or equal to the inport *firing_threshold.* The node then "contends" for its hardware resource (the attribute *execution_order* orders any queue for the resource). When the resource is available the node becomes primed and removes a number of tokens equal to the *token_consume_rate* from the arc queue at each of its input ports. After a delay equal to the attribute *firing_delay*, the node "fires" placing a number of tokens equal to the *token_produce_rate* on the arc queue at each output port. At this point the next enabled node in the hardware model queue, if any, can be primed. To initialize feedback loops or for other scheduling purposes, it is often necessary to place one or more tokens in an arc queue at the start of the simulation. This is accomplished with the input port attribute *initial_token_count.* The arc attribute *queue_size* limits the number of tokens that may exist at any time in the arc queue. If firing of a node would result in exceeding queue size on any arc for which it is the source, the node is "blocked" and may not be primed until this condition is removed.

Five attributes are employed by functional simulations, but not by GIPSIM. These attributes are listed in Table 2.4

Table 2.4. CSIM/AdaSIM Attributes

Node	Node Port	Arc
package_file_name	in_token_data_type	token_data_type
	out_token_data_type	token_units

The port and arc attributes identify "types." The node attribute *package_file_name* identifies the module source file to be used by CSIMGEN, AdaSIMGEN, HELIXGEN, and ISPGEN for simulation generation.

Table 2.5 contains a list of attributes used primarily for storage of user information. The functional simulations can be made to access these attributes as part of the simulation. For each graph, node, and arc there are four attributes provided to store a floating point number, an integer, text, or a file name.

Table 2.5. Information Attributes

Graph	Node	Arc
graph_user_float	node_user_float	arc_user_float
graph_user_text	node_user_text	arc_user_text
graph_user_integer	node_user_integer	arc_user_integer
graph_user_file_name	node_user_file_name	arc_user_file_name

Finally, one node attribute, *module_class*, assigns hardware and software nodes to user labeled classes. This attribute specifies the hardware module class to which a software node may be mapped by the automatic software to hardware mapping tool, ASH.

The second category consists of those attributes written to the database by the simulation tools. These attributes are updated by the simulation program. Node and arc attributes indicating activity or activity level can be displayed by color coding (cold to hot colors) on the graphic screen. The screen is constantly updated with these color codes during a simulation run providing an animated picture of what is going on. The program produced attributes are shown in Table 2.6.

Table 2.6. Tool Output Attributes

Node	Arc
node_utilization	current_token_count
module_utilization	average_token_count
node_latency	maximum_token_count
times_fired	token_access_count
when_next_available	
simulation_status	
status_message	

The attribute *node_utilization* gives the percent of the time units during the simulation that the node was busy. The attribute *module_utilization* is the same as *node_utilization* for a hardware graph. For a software graph *module_utilization* is the utilization of the hardware module to which the node is mapped. *Node_latency* is the earliest time that a node can finish execution. The attribute *times_fired* is the number of times a node fired during the simulation.

The attribute *when_next_available* is the time unit in the execution cycle during which the hardware node (or the hardware module associated with a software node) becomes

available. A value of zero indicates the module was never used. A value less than the current simulation time indicates the module is not in use and has been available since the time indicated. A value greater than the current time indicates the module is in use and will not become available until the time indicated.

The attribute *simulation_status* shows the node's status at the end of the simulation. This attribute may have a value of:

- INIT – node has not been blocked, primed or enabled yet.
- BLOCKED – node cannot fire (reason given by status_message attribute).
- PRIMED – node is primed.
- INACTIVE – node is set to non-firable by a user (by a procedure in CSIM or AdaSIM).
- ENABLED – node is enabled.

The *status_message* attribute is a short text message describing why a blocked node is blocked.

Four arc attributes are set by the simulation program. These four are the *current_token_count* on the arc's queue when the simulation ended; the average number of tokens on the arc's queue at any point during the simulation; the maximum number of tokens in the arc's queue at any point during the simulation; and the number of

times tokens were placed into or removed from the arc's queue by its source and sink nodes during the simulation.

3. Description: ADAS Model Of AARTS

The focus of the ADAS model was on the performance of the I/O and message passing services of AARTS, rather than the run time system itself. A complete set of design specifications was provided by TRW. Documentation was also provided for the target hardware, the Block Transfer bus, the PI-bus, and the VHSIC modules. From this documentation and discussions with TRW engineers, RTI and VPI developed an ADAS model representing the AARTS process.

This description of the ADAS model is organized into major processes before, during, and after reconfiguration, namely, (i) the startup process, (ii) the system messages component, (iii) normal operations, (iv) reconfiguration following failure of a CPU, and (v) the shutdown process.

3.1. Assumptions and Conventions

3.1.1. Assumptions

The ADAS model assumes that the Block Transfer Bus Interface Modules (BTBIMs) in each cluster will be actively loaded (loaded over the HSDB) with their entire software load. Following this, each BTBIM will support passive loading (loading over the PI-bus) of the remaining modules in their cluster. The first modules to be passively loaded are the Mission Avionic Bus Interface Modules (MABIMs). This ensures that inter-cluster communication will exist before the arbitration process

commences and prevents the likelihood of two separate system supervisors being established.

In the model, the sequence of loading AS0 into the CPU modules is controlled. This predetermines the system supervisor (CPU22). This controlled sequencing also predetermines the system *hot-backup*, (CPU12). As presently configured, the model assumes that the loading of AS0 is sequential. That is, the entire file is loaded, checksummed and started in one module before the load of the next module commences.

In the ADAS model an effort was made to balance, by size, the loading of LPUs into the CPU modules. After CPU11 fails, the three LPUs originally loaded into CPU11 are distributed over the three remaining CPU modules during the reconfiguration process. All modules, however, are loaded with AS0 (80K). Table 3.1 depicts the allocation of LPUs at system startup while Table 3.2 depicts the allocation of LPUs following system reconfiguration.

3.1.2. Node Name Conventions

The format for naming the nodes throughout the ADAS model is

NODE_NAME[instance_number]

where *instance_number* can range from 0 to the number of times that the node name appears within a single graph minus 1. If a particular node name appears only once

Table 3.1. LPU Loading During Startup

Module	LPU Size	LPU Name	LPU Designation
CPU11	13K	Navigation	LPU1
CPU11	13K	Sensor Management	LPU2
CPU11	17K	Display Generation System (DGS) Interface	LPU3
CPU12	80K	AS1	Hot Backup
CPU21	21K	Cockpit Interface	LPU1
CPU21	17K	Guidance	LPU2
CPU22	80K	AS1	System Supervisor

Table 3.2. LPU Loading Following Reconfiguration

Module	LPU Size	LPU Name	LPU Designation
CPU12	80K	AS1	Hot Backup
CPU12	13K	Sensor Management	LPU1
CPU21	17K	Guidance	LPU1
CPU21	21K	Cockpit Interface	LPU2
CPU21	17K	Display Generation System (DGS) Interface	LPU3
CPU22	80K	AS1	System Supervisor
CPU22	13K	Navigation	LPU1

within a single graph, then it will not have an *instance_number* appended to it.

Throughout the model all primitive processes are consistently named to represent the particular process they are intended to represent. For example, the node named PIBUS represents the time and resources that were utilized during actual transmission on a PI-bus. Where possible, nodes have been color-coded to indicate the type of resource they consume.

In some instances it is necessary to either (i) copy a single token to multiple sink nodes - in this case a *split* node is used, (ii) merge multiple arcs into a single arc - in which case a *join* node is used, or (iii) delay the token for a specific period of time - in which case a *delay* node is inserted between the two primary nodes.

Since *split* and *join* nodes provide flow redirection only, they do not utilize resources. The *hw_module* attributes for all *split* and *join* nodes are set to *na*, and their *firing_delay* is set to 0.0. They do not perturb the overall simulation timing.

The *delay* nodes receive a specified *firing_delay* equal to the delay between events for each event repetition they represent. For example, a process that cycles at 8Hz will have a delay representing the 0.125 seconds between consecutive executions. It is necessary that all *delay* nodes within the model have a resource available at the time the delay commences. For this reason, all the *delay* nodes are assigned a dedicated *hw_module*. These are named *delay1* ... *delayn*, where "n" is the number of *delay* nodes that appears throughout the entire ADAS model. In the simulation report

(sim.out), all statistics gathered on the *delay1* ... *delayn* resources may be ignored.

3.1.3. Primitive Hardware Components

The ADAS model of the AARTS Demonstration-3 considers the modules listed in Table 3.3 to be primitive hardware resources. Their utilization is considered in the final simulation analysis. This might be called the degree of granularity or resolution of the ADAS model. Since there are two clusters in the AARTS Demonstration System, it is necessary to make a distinction between the two. Therefore, the names of all hardware modules incorporate a cluster number as well as an optional instance number, as follows:

hw_module[cluster_number][instance_number]

Table 3.3 describes what each of the *hw_modules* used in the ADAS model represents.

3.2. Hardware Model

The top-level ADAS hardware graph of the AARTS Demonstration System (refer to Figure A-2) contains nodes representing the MAB, BTB, M1553B, Cluster1, Cluster2, System Mass Memory, Pilot Input, Sensors and Display Generation Interface.

Both Cluster1 and Cluster2 are internal nodes and expand into subgraphs. Figure A-3 is one of these subgraphs. This graph depicts the modular components of a VAMP. The graph includes the five modules, the two PI-busses, and the interfaces

Table 3.3. ADAS hw_module Names

Module	Cluster#	ADAS H/W Name
CPU 1	1	CPU11CPU
PI-bus IU Cpu 1	1	CPU11PIBIU
CPU 2	1	CPU12CPU
PI-bus IU Cpu 2	1	CPU12PIBIU
CPU 1	2	CPU21CPU
PI-bus IU Cpu 1	2	CPU21PIBIU
CPU 2	2	CPU22CPU
PI-bus IU Cpu 2	2	CPU22PIBIU
CPU in MABIM	1	MAB1CPU
PI-bus IU in MABIM	1	MAB1PIBIU
MABIU for MABIM	1	MAB1BIU
CPU in MABIM	2	MAB2CPU
PI-bus IU in MABIM	2	MAB2PIBIU
MABIU for MABIM	2	MAB2BIU
CPU in BTBIM	1	BTB1CPU
PI-bus IU BTBIM	1	BTB1PIBIU
BTB IU for BTBIM	1	BTB1BIU
CPU in BTBIM	2	BTB2CPU
PI-bus IU BTBIM	2	BTB2PIBIU
BTB IU for BTBIM	2	BTB2BIU
CPU in M1553B	1	M1553B1CPU
PI-bus IU M1553B	1	M1553B1PIBIU
M1553B Bus IU	1	M1553B1BIU
CPU in M1553B	2	M1553B2CPU
PI-bus IU M1553B	2	M1553B2PIBIU
M1553B Bus IU	2	M1553B2BIU
System Mass Memory		SMM
The Block Transfer Bus		BTB
The Mission Avionics Bus		MAB
The PI-bus	1	PI-bus1
The PI-bus	2	PI-bus2
All split and join nodes		na
All delay nodes		delay1...delayn

to the M1553B, MAB, and BTB busses (represented by graph in and out ports). Each module on this level expands into a subgraph. Figure A-4 and Figure A-5 show a CPU and Bus interface module subgraph respectively. The components in these graphs are the lowest level of resources used in the ADAS simulation.

3.3. Software Model

The basic structure of the software model is shown in the top-level ADAS software graph of the AARTS Demonstration System (Figure A-6). It is comprised of five major components: **STARTUP**, **NORMALOPERATIONS**, **SYSTEMESSAGES**, **RECONFIGURATION** and **SHUTDOWN**. The simulated flow of operation of the ADAS model is **STARTUP**, followed by **NORMALOPERATIONS**. Upon the simulated failure of CPU11, **RECONFIGURATION** is simulated while non-failed processes continue to function in the **NORMALOPERATIONS** hierarchy. This is followed by a reconfigured **NORMALOPERATIONS**. Finally, a Pilot mode change input triggers shutdown of the system. **SYSTEMESSAGES** which simulates the messages associated with management of the system commences as the AARTS software is loaded and continues for the entire operation. Nodes **SENSOR** and **PILOTINPUT** are a part of **NORMALOPERATIONS**.

3.3.1. Startup Process

The startup graph, Figure A-7, is arranged in four columns of nodes representing separate phases of the startup and configuration process. Each column contains a

node for each module involved in the particular phase. The middle row of nodes represents the activity of the SMM during each phase with cluster 2 and cluster 1 activity being represented above and below this middle row, respectively. For example, referring left to right in the middle row of Figure A-7, SMM will first load the AS0 (address state 0) software into the BTBs (node **SMMAS0TOBTB**), then load the AS0 software into the other cluster modules, (**SMMAS0CPUS**). It then loads the AS1 (address state 1) software into supervisory modules, (**SMMAS1CPUS**), and finally loads the LPUs (loadable program units) into available CPUs, (**SMMLPULOADS**). One would not normally represent this much detail on a single high-level graph. In this case, the detail is included to facilitate model understanding and demonstration.

All nodes on the STARTUP graph have subgraphs except the graph inport and outport nodes, the power-on delay nodes, the **SUROM...x** nodes for passively loaded modules, and the **ENDSTARTUP** node. The four phases of the startup process represented by the columns from left to right are:

- Execution of SUROM (Startup ROM) and BIT and active loading (loading over the BTB) of the BTBIM modules
- Passive loading (loading over the PI-bus) of the bootstrap loader and AS0 software into the remaining modules
- Arbitration for supervisory roles and loading of the AS1 software into supervisory modules

- Loading of the LPUs

Each of these phases is discussed in a separate section following this introduction.

3.3.1.1. SUROM Execution and Active Loading

The five modules per cluster each load and execute a startup ROM (SUROM) upon power up. Upon completion of the built-in test (BIT) and sensing of the appropriate discrete, each module, except the BTBIM's, moves into the second column (Figure A-7) where it transmits a ready message, while waiting to be loaded with their bootstrap loader. For the BTBIMs, after SUROM is executed, the module *actively* loads the AS0 software and the Block Transfer Bus (BTB) driver software. They then download the LPU attributes file. After completion of the download, the BTBIM moves into the second column where it loads the other modules in the cluster.

This process for a Block Transfer Bus Interface Module (BTBIM) is represented by nodes **SUROMBTBx** in the STARTUP graph. The nodes **SUROMBTBx** expand into subgraphs, which contain 4 separate functional areas (Figure A-8):

- run SUROM, initialize BTB interface, wait for token, and signal SMM that the BTBIM is ready for loading
- receive the 4-word response from SMM

- receive the AS0 from SMM in 4K blocks and run a checksum on it once the download has completed

- *open*, then *read* the LPU_ATTRIBUTES file from SMM

The ADAS graph in Figure A-8 shows that upon completion of SUROM tests (nodes **STARTSUROM**, **SETUPBTBDOWNLOAD** and **BTBBIU0**), the BTBIM transmits a 2-word message (nodes **BTB**, **XMITREADY** and **BTBIU1**) via the BTB to the the System Mass Memory (SMM) at a 10Hz frequency (controlled by node **delay**). The BTBIM receives (**RCV4WORD** and **BTBBIU2**), a 4-word message indicating the destination address, size, expected checksum, and execution start address of the BTB software that will follow from the SMM. Following receipt of the 4-word message, the BTBIM begins to receive the AS0 data in 4K word blocks (represented by nodes **RCVAS0** and **BTBBIU3**). These nodes will execute 20 times, representing the receipt of twenty blocks of data. The BTBIM runs a checksum once AS0 is loaded (node **RUNCHECKSUM**) and then proceeds to activate the subgraph of node **READLPULOC**.

Node **READLPULOC** is an *internal* node. The subgraph is shown in Figure A-9. This graph contains four functional areas:

- BTBIM transmits *open* LPU_ATTRIBUTES file request to SMM (nodes **OPENLPU-LOC**, **WAIT4TOKEN0**, **BTB0** and **XMITOPEN**)

- BTBIM receives the SMM response, then transmits a *read* request to SMM (nodes **RCVRESPONSE**, **READLPULOC WAIT4TOKEN1**, **BTB1** and **XMITREAD**)

- BTBIM receives the LPU_ATTRIBUTES file (node **RCVDATA**)

- BTBIM receives the *status* of the transmission from SMM (node **RCVSTATUS**)

3.3.1.2. SMM Response During Active Load

The node **SMMAS0TOBTB** in Figure A-7 expands to the subgraph in Figure A-10. This graph represents the SMM response to the BTBIM's transmissions. The SMM first loads the BTBIM software into BTBIM2, the right half of the graph, and then into BTBIM1, the left half of the graph. In Figure A-10, the node **WAIT4TOKEN2** represents the token-wait time before SMM can transmit the 4-word message to BTBIM2. The actual transmission of the 4-word message is represented by node **XMITDATA1**. The loop consisting of nodes **WAIT4TOKEN3**, **BTB1**, **split3** and **delay** represents the token-wait time, utilization of bus resources, a split to send off tokens to different recipients, and an interblock delay respectively during the twenty consecutive 4-kilobyte data transmissions of AS0 to BTBIM2. Node **WAIT4TOKEN** delays while the token is received from the preceding station on the ring. Since the guidance was to assume serial loading of the BTBIMs, this delay represents the token passage around the ring to the predecessor node. Following completion of the AS0 load to BTBIM2, the SMM responds to the *open* and *read* LPU_ATTRIBUTES file requests from BTBIM2. This response is simulated by the subgraph of node **RDLPULOC1**, (Figure A-11). The two principal areas of *smm_rdlpuattribs.swg* are: receive the *open* request and transmit a response; receive a *read* request and transmit the LPU_ATTRIBUTES file, followed

by a read status.

After BTBIM2 receives the AS0 download, the SMM commences to load BTBIM1. The arc connecting node **split3** to **WAI 'OKEN0** in Figure A-10 is used to signal the completion of the BTBIM2 load and initiate the BTBIM1 load.

3.3.1.3. SUROM and Passive Loading of AS0

The non-BTBIM modules, upon completion of SUROM tests, each transmits a 2-word message on the PI-bus at a 1Hz interval. This 2-word message contains the address and module type. It represents a repeating request to the BTBIM for a download of its bootstrap loader. The SUROM nodes for non-BTBIM modules (Figure A-7) are *leaf* nodes and are assigned a *firing_delay* representing the entire SUROM processing. The BTBIM receives the 2-word message on the PI-bus. It then determines and then transmits the correct 4-word message to the requesting module. This 4-word message consists of word count, data destination address, execution start address, and expected word count. The BTBIM will subsequently broker the download of the bootstrap loader from the SMM (via the BTB) to the requesting module (via the PI-bus). Three separate ADAS graphs hierarchies are required to depict the following:

- the requesting module's activity during AS0 load
- the dual communication maintained by the BTBIM - communicating with SMM, via the BTB, while simultaneously communicating with the requesting

module, via the PI-bus

- the SMM role during the non-BTBIM module load of the bootstrap loader and AS0

The loading of the bootstrap loader and AS0 is the same for all non-BTBIM modules. The loading of CPU22 by BTBIM2 will be discussed in detail.

3.3.1.4. CPU Role During Passive Load

Figure A-12 is the graph of node **AS0CPU22** on the startup graph (Figure A-7), which represents the requesting CPU processes during the passive loading. This graph simulates the following functions:

- Upon successful completion of BIT, the module commences periodic transmission of the 2-word message (left column)
- The module then receives the 4-word message and sets up for download (2nd column)
- After receipt of the bootstrap loader, the module opens the AS0 file (3rd column)
- Upon receipt of the open file response (node **RCVAS0RESP**), a read request is transmitted. This is followed by a data block, (node **RCVAS0DATA**) and then a

read status message (again on node **RCVASORESP**) which initiates another read. This continues until the last block of AS0 is received.

- Upon receipt of the last read status for the AS0 file, a checksum is run, (node **CHECKSUMAS0**). After a successful checksum the AS0 software is started (the outport starts appropriate system messages).
- After AS0 is started, the module opens, reads, and checksums the LPU attributes file and passes into the arbitration column of the parent graph.

3.3.1.5. BTBIM Role During Passive Load

Node **AS0BTB2** in Figure A-7 is an *internal* node. Its subgraph (Figure A-13) contains four *internal* nodes representing the other four modules in the cluster. All four nodes have the same subgraph hierarchy. Each of these nodes expands into the subgraph shown in Figure A-14. At this level the two distinct communication channels for BTBIM2 are represented by *internal* nodes **BTBTOSMMAS0** (to the SMM), and **BTBTOCPUAS0** (to the CPU).

Figure A-15 is the expansion of node **BTBTOSMMAS0**. It represents the flow to the SMM. There are three processes represented on this graph.

- On receipt of the two-word message from the CPU, the BTBIM builds the appropriate 4-word response and transmits it to the CPU. Actually, the token

travels over the arc between the two nodes on the parent graph. Transmission to the CPU module is represented in the graph discussed in the next paragraph.

- Along with transmission of the 4-word response, the BTBIM transmits an open file message for the bootstrap loader to the SMM. Upon receipt of the open file response from the SMM, the BTBIM transmits a read message to the SMM. All subsequent responses from the SMM are acted on in the subgraph discussed in the next section.

- Following receipt of the bootstrap loader, the CPU issues open and repeated read commands to obtain its load. Forwarding of these commands to the SMM is represented by the left-hand column of the graph.

The subgraph of node **BTBTOCPUAS0** (Figure A-16) represents the following sequential BTBIM processes.

- Once the correct 4-word message for the CPU has been determined, the BTBIM transmits the message to the CPU (left column)

- After the read command has been issued for the bootstrap loader, the BTBIM receives the file from the SMM and transmits it to the CPU via the PI-bus (nodes **RCVBOOT, XMITDATA0, CCBEXECUTION2, PIBUS2** and **BTBPIBIU2**). This is followed by a read status message (node **RCVBOOTRESP**)

- The column headed by node **RCVFILERESP** represents the passage of the open

file response and read status messages as the CPU reads the AS0 software in 4-k blocks. The column headed by node RCVAS0 represents the passage of the data blocks. These two columns alternate as tokens are received from memory until AS0 download is complete.

- When the SMM responds to an *open* LPU_ATTRIBUTES file request by CPU22, the BTBIM transmits first the response (node RCVATTRIBRESP) then the LPU_ATTRIBUTES file to the CPU (node RCVATTRIB). Finally, the read status is forwarded, again through node RCVATTRIBRESP.

3.3.1.6. SMM Role During Passive Load

Figure A-17 is the subgraph of node SMMAS0CPUS on the startup graph (Figure A-7). The graph inports and outports represent tokens arriving from and passing to the appropriate BTBIM. The internal arcs enforce the policy of sequential loading of the modules. All nodes except the first and last on Figure A-17 expand into the same subgraph which is shown in Figure A-18.

On Figure A-18, nodes inport0 and outport1 are the connections to the preceding and following nodes respectively of the parent graph (the internal arcs on that graph). Nodes inport1 and outport0 carry the tokens between the SMM and the BTBIM. This graph contains six primary functions:

- The left column executes when the BTBIM requests opening of the bootstrap

loader file. It returns the open file response to the BTBIM.

- Upon receipt of the read bootstrap loader message, the second column is executed passing the file followed by a read status message.

- The center column receives the open file requests for both the AS0 and for the LPU attributes file and returns an open file response (it is a duplicate of the first column).

- The reads for each 4K block of AS0 enter the fourth column and follow a path over to and through the second column. This takes advantage of the fact that the bootstrap loader is also a 4K block.

- The remainder of the 4th column is inactive in the current simulation. It would simulate transfer of the final short block of a file that does not divide exactly into 4K blocks. The additional column would be needed to account for the different bus and interface utilization of the shorter message. This column has been included in the current model to provide flexibility and to demonstrate how similar columns would be placed in the BTBIM and CPU portions of the model to simulate loading a file of a different size.

- The final column represents the transmission of the LPU_ATTRIBUTES file followed by a read response. Again, it is a duplicate of other columns except for the resource usage simulated.

3.3.1.7. Arbitration

The arbitration process follows completion of AS0 loading. CPU22 is the first CPU to receive and execute the AS0 software, and it will (as the model is currently configured) become the system supervisor module. The subgraph of node ARBCPU22 on the startup graph (Figure A-7) is shown in Figure A-19. All four process nodes on this graph have subgraphs. Some will not be described in detail. Figure A-19 depicts the four stages of arbitration:

- Node ARBCLUSTER represents the issuance of five cluster supervisor heartbeats. The feedback loop containing the delay controls the timing of the heartbeat.
- After "winning" cluster supervisor arbitration node LOADAS1 is activated. The subgraph of this node is similar to the one for loading AS0 (with the 2- and 4-word messages, the bootstrap loader, and the loading of the LPU_ATTRIBUTES file deleted). This is supported by analogous BTBIM and SMM hierarchies.
- Upon starting the AS1 software, the hierarchy below node ARBHB commences execution. The subgraph hierarchy of this node is similar to node ARBCLUSTER except that heartbeats are transmitted on the MAB and provision is made to stop the hot backup heartbeat upon assumption of the system supervisor role (the feedback loop from node ASSUMESYSSUP).
- After 5 successful hot backup heartbeats, node ASSUMESYSSUP is activated and five system supervisor heartbeats are issued. When this is completed, the hot

backup heartbeat is stopped, a token is passed to the system messages function to commence the supervisory messages, and a token is passed to node **LPUCPU22** on the startup graph.

CPU12, in the meantime, assumes the role of cluster1 supervisor and once it detects the absence of the *hot-backup* pulse, which ceased when CPU22 became system supervisor, it (CPU12) assumes the role of *hot-backup*. The token passed on node **tocpu12** initiates hot-backup arbitration in CPU12.

3.3.1.8. Loading of LPUs

After completion of the arbitration process, the simulation proceeds to the loading of the LPUs, the right hand column of the startup graph (Figure A-7). This is triggered by the transmission of a CONFIG_REQUEST message by the system supervisor, CPU22. The individual *cluster supervisors* relay the CONFIG_REQUEST to the other CPUs within the cluster via the PI-bus. The loading of individual LPUs follows in a manner similar to the loading of AS0 and AS1, with the BTBIM brokering all transfers from SMM to the CPU.

Figure A-20 is the subgraph of node **LPUCPU22**, the system supervisor node. The left two columns plus node **RCVDATA** at the top of column 3 simulate downloading the mission database file. The remainder of column 3 and lower portion of column 4 simulate the generation and issuance of the configuration commands. Node **CLUSTER1ACTIVE**

is the connection to node **CL1ACTIVE** on the parent graph. This connection prevents configuration from starting until all modules have loaded and started the AS0 Software. The upper portion of column 4 simulates receipt of the configuration reports upon completion of the LPU loads.

Node PI-bus0 *through* PI-bus3 *all have subgraphs that look like* Figure A-21. One graph like that in Figure A-21 has been prepared for each pair of modules that exchange PI-bus messages (the hardware modules on which the nodes are to be mapped) and for each size of message exchanged (the amount of resource used). This graph is referenced in the node attribute *subgraph_filename* for each exchange of a message of that size between those two modules, and a copy is incorporated into the flattened graph upon starting the GIPSIM simulator. This ability to share graphs in an ADAS model helps control the size of the model database. This feature was seen earlier in this report where the same hierarchy is used four times in the BTBIM portion of the AS0 load and the same graph used six times in the SMM portion. One could employ a single graph file like Figure A-21 for all message exchanges and use a script file on startup of the simulation (GIPSIM, CSIM or AdaSIM) to assign the hardware mapping and firing delay for the nodes in the various instances in the model.

Figure A-22 shows a CPUs response to the configuration request message. In this case, two LPUs are loaded and a status report is returned to the cluster supervisor. The CPU portion of the loading of an LPU is represented by the subgraph of node **READ1PU1** shown in Figure A-23. This is similar to the loading of AS0 shown in

Figure A-12 with the 2- and 4-word messages, the boot, and the LPU attributes file deleted. The CPU issues an open file message, left column; receives a response and issues a read (second column); and then alternates between node **RCVDATA0** and the second column as the 4K blocks are received. The final short block executes node **RCVDATA1**. Receipt of the final read status message initiates node **RUNCHECKSUM**.

Figure A-24 is the subgraph of node **BTB2LPULOADS** on the startup graph. It contains an internal node for loading the mission database into the system supervisor and one for each of the two LPUs to be loaded into CPU21. Figure A-25 is the subgraph of node **CP21READ1PU1**. From left to right the columns pass on the CPU open and read messages to the SMM, the SMM response messages to the CPU, the 4K data blocks to the CPU, and the short data block to the CPU.

Figure A-26 is the subgraph of node **SMMLPULOADS** on the startup graph. It contains an internal node for each LPU. For like size LPUs, a single subgraph file has been used. Figure A-27 is the subgraph of node **LPU1CPU21**. From left to right the columns respond to the open message, respond to the read requests for 4K blocks, and respond to the read request for the short block.

Figure A-28 is the subgraph of node **LPULDMAB2** on the startup graph. This graph simulates the MABIM in cluster 2 receiving the configuration request message from the system supervisor (bottom of third column in Figure A-20), and transmitting it to the MABIM in cluster 1. The token entering this graph on node **inport** is the

token that was passed upon completion of the MABIMs loading of AS0 (see arc on Figure A-7. The token entering on node **fromCP22** is the one passed from node **LPUCPU22** to node **LPULPMAB2** on Figure A-7. Node **MAB** is an internal node. Its subgraph, Figure A-29, is similar to and employed in the same way as the PI-bus subgraph (Figure A-21).

Figure A-30 is the subgraph of node **LPULDMAB1** on the startup graph. It receives the token from node **join** in Figure A-29 and transmits it on the PI-bus to the cluster supervisor in cluster 1 (CPU12). Node **PIBUS** has a subgraph like that in Figure A-21.

3.3.2. System Messages

Returning to the top level graph (Figure A-6), node **SYSTEMESSAGES** contains the graphs that simulate the establishment of connections for and passing of commands, reports, and pings or pulses necessary to manage the system (except for the actual passage of configuration or shutdown request and report messages which have been modeled in these startup, reconfigure, and shutdown portions of the model).

Figure A-31 is the subgraph of node **SYSTEMESSAGES**. The eight nodes at the bottom contain the activity of the eight specialist modules. These nodes are started upon receipt of a token on the graph inports **as0startedx** and terminate with a token on graph inport **startreconfig** (actually should be named "failure") for node **CPU11** and **fromShutdownx** for the others. The two nodes at the top contain the activity of the supervisory modules. The supervisory module nodes have an additional set of functions that start

upon loading of the AS1 software.

Figure A-32 is the subgraph of node CPU11 on the system messages graph. Upon starting of the AS0 software, various message receive and transmit connections are made. This is shown on the upper portion of the graph. Upon completion of the connection for the ping message, the module starts periodic transmission of the ping (I'm ready) on the PI-bus. This is simulated in the left hand loop with a hierarchy below node XMITCLUSPING. When the ping is acknowledged by the cluster supervisor inport fromCPU12, the ping is halted (a series of inport tokens from node split1 to node OR0 "chokes" the loop) and the module pulse (heartbeat) is started. Inport Stop terminates the pulse on shutdown in the same way the ping is terminated. We will describe the simulation of the transmission (nodes XMITCLUSPING and XMITMODPULSE) in the description of the u/c transmission of supervisor heartbeats. One additional feature of the model is shown by the two nodes below node XMITMODPULSE on this graph. For messages that do not elicit a response, the entire process, including receipt of the message by the addressee, is modeled in the subgraph of the initiating process. This reduces the number of arcs on the higher level graphs.

Figure A-33 is the subgraph of node CPU22 in the system messages graph (Figure A-31). The top portion is the establishment of message connections upon starting the AS0 software. The initial pings and pulses for the supervisory module are modeled in arbitration graphs in the startup hierarchy. The bottom portion of the graph is started when the AS1 software is started in the cluster supervisor. A series of connec-

tions are made. These, since they are system-wide connections, require transmission of the channel array to the MABIM (nodes OR, XMITARRAY modules and MABUPDATEARRAY). Node XMITARRAY has a subgraph hierarchy similar to those shown in Figure A-28 and A-29 with the transmission being on the PI-bus rather than the MAB.

Node CLPINGACK receives and acknowledges the pings from the other modules in the cluster. Its subgraph is shown in Figure A-34. Once the connections are established (top two nodes), each ping receives a response. This, it turns out, is immediate for the CPU, MABIM and BTBIM modules since they are pinging before the AS1 software initial reading suggests AS0 is started is started in the cluster supervisor. The M1553B module receives a response to its first ping. It does not complete the AS0 load until after the AS1 software is started in the supervisor model. The four nodes, PIBUS2x, each have a PI-bus transmission subgraph.

Figure A-35 is the subgraph of node CLPULSES on the System supervisor graph. This figure combines the system supervisor (left column) and cluster supervisor heartbeats. The loops are similar to those for the ping and pulse on the specialist node graph (Figure A-32) with the transmissions being stopped during shutdown by nodes inport1 and split0. After the message connections have been established, the pulse transmission is commenced.

Figure A-36 is the subgraph of node XMITPIBCLPLS. This same graph (nodes mapped

on different hardware) is found below the xmit ping and xmit pulse nodes on the specialist subgraphs. Node **PIBUS**, of course, has a transmission subgraph. In this case, since there are multiple addressees, it is different from the point-to-point one. This graph is shown in Figure A-37. Node **XMITHSDBCLPLS** in Figure A-35 transmits the pulse to the MABIM. **XMITHSDBCLPLS** has a subgraph like Figure A-36 with the PI-bus sub-subgraph shown in Figure A-38. This differs from Figure A-21 only in having a second outport that feeds the loop for the repeated message. Node **MAB2** on Figure A-35 transmits the pulse to the MABIM in the other cluster. The hierarchy below node **MAB2** is the same (except for a single graph inport) as that shown for the configuration request in Figures A-28 and A-29. Similarly, the hierarchy below node **MAB1** which transmits the message to CPU12 (the hot-backup) is equivalent to that shown in Figure A-30 with a PI-bus transmission subgraph.

3.3.3. Normal Operations

During normal operations the system messages and LPUs are running with LPUs receiving inputs from the pilot and the sensors. Figure A-39 is a data flow diagram of the demonstration three applications. Referring to the AARTS top-level graph (Figure A-6) during normal operations, nodes **SYSTEMESSAGES**, which commenced during startup; **NORMALOPERATIONS**, whose execution is triggered by completion of startup; **SENSORS**; and **PILOTINPUT** are executing.

Figure A-40 is the graph of node **SENSORS**. The simulation merely places the sens-

ings, AirData at 5 Hz and INSDATA at 32 Hz, on the MAB. It is pulled off at the appropriate frequency by the appropriate subgraph of node **NORMALOPERATIONS**. The graph port on the left starts the execution and that on the right terminates it upon shutdown. The graph for node **PILOTINPUT** is structured like Figure A-40 with the two columns representing the transmission of the MMKey (mission mode) and the IMFKey (multifunction display). In addition, since each input solicits a response from the cockpit interface LPU, a graph outport is included below each **MABx** node.

Figure A-41 is an expansion of node **NORMALOPERATIONS**. The left half contains the graph inports and nodes to distribute their tokens. The right half contains two columns of internal nodes and the graph outports. Upon completion of startup, a token is received on node **start3** and distributed by node **split** to the five internal nodes in the left column. These tokens start simulated execution of the LPUs. Node **CPINTERFACE** emits a token to node **PILOTINPUT** on the parent graph when it has established connections and then receives inputs from **PILOTINPUT** through graph ports MMKey and IMFKey. Similarly, node **SENSORMGMT0** turns on node **SENSORS**. When the simulated failure occurs a token is received via node **failure**, and node **blockCPU11** emits tokens to stop the processing in nodes **NAVIGATION0**, **SENSORMGMT0**, and **DGSINTERFACE0**. After the failure is detected and the LPUs have been loaded into their new CPUs, nodes **NAVIGATION1**, **SENSORMGMT1**, and **DGSINTERFACE1** are started (graph ports **Start0**, **1**, and **2**). A MMKey input (mode change) into node **CPINTERFACE** causes an output on graph port **shutdown** that triggers the shutdown

process. As LPUs are stopped in the shutdown process, tokens are received on nodes **shutdown0** through **shutdown3** stopping execution. The execution of node **CPINTERFACE** is stopped by stopping node **PILOTINPUT** on the parent graph.

Figure A-42 is the subgraph of node **CPINTERFACE**. Upon receipt of the "start" token on node **inport0**, message connections are established (left column). Node **outport1** initiates node **PILOTINPUT** on the top-level graph. The LPU then responds to inputs from the pilot (nodes **rcvmmp** and **rcvimfk**) and **waypoint** changes (node **inport1**) from node **GUIDANCE** on the parent graph. Changes in the variable "FLYTO" received or derived from **PILOTINPUT** are output to node **GUIDANCE** on the parent graph (node **outport0**). After an appropriate number of MMP inputs, a token is emitted on node **outport2** to initiate system shutdown. It can be seen from this figure, and those that follow, that the focus of this model is on data transfer which is modeled in much greater detail than the CPU processing.

Figure A-43 is the subgraph of node **SENSORMGMT0**. As with **CPINTERFACE**, connections are made first with establishment of appropriate connections enabling processing of messages. Outputs are provided to the Guidance LPU in CPU21 over the MAB and to the Display Generation System (DGS) Interface and the Navigation LPUs internally. Figure A-44 is the subgraph of node **SENSORMGMT1**. After the failure, this LPU is loaded into CPU12, Navigation has been reloaded into CPU22, and DGS interface into CPU21. As can be seen, the changes from Figure A-43 are primarily the path for the outports. The only other difference is the deletion of the graph outport

that turned the sensors on. Figures A-45 through A-47 are one version each of the other three LPUs.

3.3.4. Failure and Reconfiguration

Figure A-48 is the subgraph of node DETECT on the top-level graph (Figure A-6). This graph is initiated by expiration of the firing delay of node FAILURE (a leaf node) on the top-level graph. Node DETECTMISNGPLSE delays for 10 pulse periods which is the specified number (ref 11) for the cluster supervisor to take action. The remainder of the graph represents the transmission of the configuration report (failure) message from the cluster 1 supervisor to the system supervisor.

Figure A-49 is the subgraph of node RECONFIGURE on the top-level graph. The upper portion consists of nodes associated with preparation and transmission of the configuration requests messages necessary to load and start the failed LPUs in new modules. The last two nodes, other than graph ports, in each column are internal nodes that load and start the assigned LPU. Below each ...LOADLPU node is a subgraph with three internal nodes representing the CPU, BTBIM, and SMM functioning in the loading process, (Figure A-50). Below each node in Figure A-50 is a subgraph of the form we have seen in Figures A-23, A-25, and A-27. In fact, where the module and the size of the LPU is the same, the same graph is used. The nodes ...STARTLPU on Figure A-49 have a subgraph that starts the LPU sending a token into the appropriate subgraph of NORMALOPS. They then transmit the configuration report to

the system supervisor. Figure A-51 is the subgraph of node CPU12STARTLPU. This configuration report must pass over the MAB to cluster 2. The other two subgraphs have only the portion necessary to transmit the report internally.

3.3.5. Shutdown

After the reconfigured system has run for the desired length of time, Shutdown is executed by passing a token from node XMITMISSMODE in Figure A-42 through the graph outport into node SHUTDOWN on Figure A-6. Figure A-52 is the graph of node SHUTDOWN. The timing can be controlled by adjusting the rate at which MMKey tokens are passed to the LPU CPINTERFACE (currently set at 1Hz) and the firing threshold attribute (currently set at 5) on node CPU22RcvShutDn in Figure A-52.

In order to simulate an orderly shutdown of the system, the model forces a sequence on the shutdown process. The module containing the system supervisor is the last module to shut down. The last module prior to the system supervisor is the MABIM module in the cluster containing the system supervisor. For other clusters, the MABIM module is the last and the cluster supervisor the next to last. If we had more than two clusters we would have assured that the hot-backup was the last of the other cluster supervisors. In Figure A-52 there are two principal internal nodes, one for the shutdown of each cluster.

Figure A-53 is the subgraph of node Cluster2ShutDn. Node CPU22Broadcast has a hierarchy below it that passes the shutdown message token to each of the other modules

in the cluster. Node **MAB2PIBU** has an inport that will prevent its execution until the shutdown report is received from cluster 1. The system supervisor, CPU22, shuts down after receiving the MAB shutdown report. Each column shows a module first stopping and then unloading any LPUs. This is followed by transmission of a shutdown report and finally stopping the module. As LPUs are stopped tokens are output to stop their execution. Nodes **output0** through **4** pass the tokens that stop the appropriate sub hierarchy in SYSTEMESSAGES. As the model is currently configured, the shutdown message to cluster 1 is transmitted after the three modules other than MAB and system supervisor in cluster 2 have shut down. If we want it to be done simultaneously, node **Xmitcl1ShutDn** (and the hierarchy below it that transmits the message on the MAB) would be moved up to the parent graph. The outports and arcs connecting the three shutdown report nodes with node **join** would be deleted as would nodes **join** and **ToCluster1**. On the parent graph (Figure A-52) the new node, **XmitCl1ShutDn**, would be placed on the graph and its inport connected to an outport that would be added to node **CPU22ComputeCnfg**. The outport connecting node **Cluster2ShutDn** to node **Cluster1ShutDn** would be deleted and a new arc connecting the outport of node **XmitCl1ShutDn** to that inport (of node **Cluster1ShutDn**) would be added. The subgraph hierarchy would automatically follow with node **XmitCl1ShutDn** by including the appropriate subgraph filename.

Figure A-54 is the subgraph of node **CPU21StopLPU**. It contains a node for each LPU. Figure A-55 is the subgraph of one of these nodes. First, the LPU is stopped. This is

followed by a series of disconnects from the LPUs messages and finally an updating of the LPU loaded and running arrays in AS0.

Figure A-56 is the subgraph of node Cluster1ShutDn on the shutdown graph (Figure A-52). The shutdown message comes into node RCVCl1Shutdown (Figure A-57) where it is transmitted to all modules on the PI-bus. This node also collects the tokens signifying that the M1553BIM, BTBIM, and CPU have completed shutdown to trigger shutdown of the MABIM. The remaining nodes function the same as those in cluster 2 shutdown. When all modules in cluster 1 have completed shutdown, node join passes a token back to node Cluster1ShutDn on the shutdown graph (Figure A-52) to trigger execution of shutdown of the MABIM in that cluster followed by the CPU22.

3.4. Resource Utilization

The GIPSIM simulation tool simulates resource utilization through the node attribute *firing_delay* assigned to the leaf nodes of the model. When a node is primed (all of its input conditions (firing thresholds) are met and its resource module available and assigned), the assigned resource module becomes unavailable for the duration of the assigned firing delay (and that period is added to the "module utilization" collected for output). Upon expiration of the firing delay, appropriate tokens are placed in the outport arc queues and the resource module is released. Thus, all firing delays must be calculated in common units. In this case the unit selected was microseconds.

The objective of this modeling effort is to analyze data transfer during configuration and reconfiguration. Those processes associated with data transfer are represented with a high degree of resolution and their delays calculated using all available data after an exhaustive search. Other processes, such as LPU functioning and configuration algorithms, were resolved only to the degree necessary to provide the appropriate profile of data transfer resource demands. This aggregation serves to reduce the running time required for analyses.

This section will describe the firing delays assigned in the model and how they were calculated.

3.4.1. Data Transmission Delays

Data transmission delays depend upon the characteristics of the transmission system and the amount of data transmitted. The following paragraphs address message sizes and PI-bus and HSDB transmission.

3.4.1.1. Message Sizes

Data transmissions were addressed in four categories.

Those messages involved in system management (transmitted and received by the AARTS Software) are described in the documentation of the system executive [11]. Table 3.4 contains a list of these messages, their origin and destination, and the

message length in words.

Table 3.4. System Management Messages

SYSTEM MESSAGES (Reference 11)							
Sender	Receiver	Name	Size	Page Number Fig.	Page Number Discussion	Bus	Comment
DE.Module_Mgr	SE.Clus_Mg	Mod_Config_Rpt	4+5/lpu	7,8	2,26	PI	
DE.Module_Mgr	SE.Clus_Mgt	Mod_Status_Rpt	3	9	2,26	PI	
PVI	SE.SS	Msn_Mode_Chg_Req	1	11	10,42	HSDB	SS incl HB
SMM(file)	SE.Msn_Ctrl	Msn_Control_File	2000	14	13	HSDB	BTB
SMM(file)	SE.LPU_Attr	LPU_Attributes_File	42	15	13	HSDB	BTB
SE.Kernel(ARB)	SE.Kernel	Cluster_Pulse	2		20,25,26	PI	Cluster Heartbeat
SE.Kernel	SE.Clus_Mgt	Cluster_Ping	2	60	20,25,26	PI	
SE.Kernel	SE.Clus_Mgt	Module_Pulse	2		20,25,26	PI	
SE.Clus_Mgt	SE.SS	Cluster_Config_Rpt	20+5/lpu	7,8	37,42	HSDB	SS incl HB
SE.Clus_Mgt	SE.Kernel	Cluster_Ping_Ack	2	61	37,20	PI	
SE.Clus_Mgt	SE.Kernel&SS	Cluster_Pulse	2		37,20,42	PI&HSDB	SS incl HB
SE.Clus_Mgt	DE.Module_Mgr	Cluster_Status_Req	2	6	37,1,42	PI	
SE.Clus_Mgt	SE.SS	Cluster_Status_Rpt	9	57	37	HSDB	SS incl HB
SE.Clus_Mgt	SE.SS	System_Ping	2	60	37,42	HSDB	SS incl HB
SE.Clus_Mgt	DE.Module_Mgr	Cluster_Config_Req	3+1/lpu	4,5	38,1	PI	
SE.Clus_Mgt(HB)	otherClus_Mgt	HB_Heartbeat	2	56	38,26	HSDB	
SE.Clus_Mgt(SS)	SE.HB	SS_Heartbeat	2	56	38	HSDB	
SE.Clus_Mgt(SS)	otherClus_Mgt	SS_Ping	2	60	38	HSDB	
SE.SS	PVI	Msn_Mode_Chg_Resp	1	12	48,10	HSDB	
SE.SS	SE.Clus_Mgt	System_Acknowledge	2	61	48,27	HSDB	
SE.SS	SE.Clus_Mgt	System_Config_Req	3+1/lpu	4,5	48,26	PI&HSDB	
SE.SS	SE.Clus_Mgt	System_Status_Req	1	59	49,30	PI&HSDB	Sys_Query
SE.SS	SE.HB	System_Status_Rpt	3+8/cl	58	49	HSDB	

Messages transmitted to and from the LPUs in the simulation were described in data provided by the TRW project manager [24]. Table 3.5 describes these messages.

Messages associated with obtaining files services are documented in the IRS [8] [9] and from data provided by the TRW project manager [23] . Those employed in the model are listed in Table 3.6.

File sizes were obtained from the TRW project in reference 24 and through telephone conversations. The file sizes used in the model are shown in Table 3.7.

Table 3.5. Messages Transmitted to and from LPUs

LPU MESSAGES (Reference 24)				
Sender	Receiver	Name	Size (words)	Bus
SENSORS	SENSOR MGT	Air Data Sensor	4	MAB
SENSORS	SENSOR MGT	INS Sensor	29	MAB
SENSOR MGT	DGS Interface	Air Data	20	MAB
SENSOR MGT	GUIDANCE	Air Data	20	MAB
SENSOR MGT	DGS Interface	INS Data	46	MAB
SENSOR MGT	GUIDANCE	INS Data	46	MAB
SENSOR MGT	NAVIGATION	INS Data	46	MAB
PILOT	PVI	IMFK Key	2	MAB
PILOT	PVI	MMP Key	2	MAB
PVI	PILOT	MMP Lamp	1	MAB
PVI	PILOT	Pilot Mission Mode	1	MAB
PVI	GUIDANCE	Fly To	1	MAB
PVI	PILOT	IMFK Command	32	MAB
NAVIGATION	GUIDANCE	Sys Body Nav State	40	MAB
NAVIGATION	DGS Interface	Sys Body Nav State	40	MAB
NAVIGATION	GUIDANCE	Nav State	17	MAB
NAVIGATION	DGS Interface	Nav State	17	MAB
GUIDANCE	DGS Interface	Steering Control	18	MAB
GUIDANCE	STORE	Steering Control	18	BTB
GUIDANCE	DGS Interface	Steer	18	MAB
GUIDANCE	PVI	Guidance Waypoint	1	MAB
DGS Interface	M1553B	Master Mode Buffer_1	32	M1553B
DGS Interface	M1553B	Master Mode Buffer_2	32	M1553B
DGS Interface	M1553B	Miscellaneous Msg	4	M1553B

Table 3.6. Files Services Message Lengths

FILES SERVICES MESSAGES (Reference 8,9, & 23)	
Service	Message Size (words)
Open File	3
Response	2
Read File	6
Read Response	2

Table 3.7. Files Services File Sizes

FILE SIZES (Reference 24 & Verbal)		
AS0	80K words	20 blocks
AS1	80K words	20 blocks
Cockpit Interface	21K words	5 blocks plus a short block
DGS Interface	17K words	4 blocks plus a short block
Guidance	17K words	4 blocks plus a short block
Sensor Management	13K words	3 blocks plus a short block
Navigation	13K words	3 blocks plus a short block
Bootstrap Loader	4K words	
Mission Control File	2K words	
LPU Attributes File	42K words	

3.4.1.2. PI-bus Transmission Delays

The following conditions and assumptions were used for determining PI-bus data transmission rates for the ADAS model.

- PI-bus is a Type 16 ED Bus [1] (page B-16)
- All modules will vie for PI-bus before every transmission
- VIE sequence takes 8 bus cycles [1] (page B-30)
- The HZ and DZ cycles have been included, implying non-transfer wait cycle before the Header and Data Acknowledge cycles [1] (Table 5-17, page B-71)
- Data will transfer from master to slave without interruption provided the amount of data transferred is less than 65,536 words
- There will be no passing of Tenure during message passing or AS0 loading for all the modules, and that the absolute tenure limit of $(2^{24}+8)$ will not be necessary
- No Parameter Write type messages were modeled
- All data transmission on the PI-bus shall use the the Block Message-SH sequence [1] (page B-69)
- The BTBIM has the highest priority (4095) of all modules connected to the PI-bus [1] (page B-110)

- The PI-bus will have a transfer rate of 10 million words per second during the DATA state, and assuming 16 bit words only, and one cycle only to transfer each word, each cycle time is 100 ns, or 0.1 microseconds
- The Block Message-SH Sequence contains the status words listed in Table 3.8 for a total overhead of 8 words [1] (page B-35)

Table 3.8. PI-bus Status Words

Message	Size (words)
HEADER (HO, HWA, HWC0, HWC1, HZ)	5
HEADER ACKNOWLEDGE	1
DATA ACKNOWLEDGE (DZ, DA0)	2

By including the necessary 8 VIE cycles and assuming 1 cycle/word, the total overhead is 16 bus cycles per transmission. This coupled with the 0.1 microsecond per word transmission rate results in the following formula:

```
Firing Delay = (#words + overhead)  x rate microseconds
             = (#words + 16) x 0.1 microseconds
```

Firing delays for common message sizes are shown in Table 3.9 These are the delays placed on the leaf node names **PIBIU & PIBUS** in the transmission graphs. It was determined during the modeling that the transfer rate between the PI-bus interface unit and the memory was slower than the bus transfer rate. This could lead to loss

of portions of messages. The modeling team was instructed to assume a two-channel transfer (2 words in parallel) between PIBIU and memory and a buffer in the PIBIU that would receive words prior to transmission and collect them on the receiving end. The initial read was modeled along with CCB execution in nodes CCBEXECUTION. The final writes were modeled along with reaction to message labels in nodes RCVxx in the model.

3.4.1.3. High Speed Data Bus (HSDB) Transmission Delays

The following conditions and assumptions were used for the calculation of the High-Speed Data Bus data transmission times.

- The Avionics Bus Interface (ABI) component of the HSDBIM is in the ACTIVE state following completion of its SUROM
- the HSDB-ABIs of both clusters remain in the network. Ring admittance is not modeled
- Transmission Monitor Timer (TMT), Transmission Streaming Timer (TST), Initialization Sequence Try Count (ISTC), Valid Message Transmitted Count, Message Echo Error Count and all "counts" listed on page 114, [2] are not modeled
- The ABI is capable of transmitting/receiving on the HSDB while simultaneously transmitting/receiving to the 1750 Module

Table 3.9. PI-bus Firing Delay

Size (words)	PI-bus (microseconds)
1	1.70
2	1.80
3	1.90
4	2.00
6	2.20
8	2.40
9	2.50
17	3.30
18	3.40
20	3.60
22	3.80
23	3.90
28	4.40
29	4.50
30	4.60
32	4.80
40	5.60
45	6.10
46	6.20
55	7.10
60	7.60
288	30.40
520	53.60
616	63.20
4096	411.20

- The ABI can fully enqueue and validate any message received before transferring it to the 1750 module
- All transfer of data (HCCB's, message data) between the ABI and the 1750 module is via DMA
- All 3 Transmit Queues on the ABI can store up to 3839 16-bit words each (one buffer for each priority level) [2] (page 103). Only priority 1 is simulated
- The ABI Receive Queue (data from HSDB, enroute to 1750 module) can store up to 8192 16-bit words [2] (page 104)
- Block size is 256 words on the MAB and 4K words on the BTB
- The Bus transfer rate is 50 million bits per second, or 0.02 microseconds per bit [2] (page 59)
- The token frame and preamble consists of 40 bits [2](page 78). See Table 3.10.
- Message frame overhead is 88 bits [2](page 78). See Table 3.11.
- An intertransmission gap of 280 nanoseconds, [2] (page 78), was used
- When the Bus is idle, the token is equally likely to be at any point on the ring.

The assumption that the token is equally likely to be anywhere on the ring at any time implies that on the average when a station is ready to transmit, it will wait for one token passage (the average of the token being 0, 1, or 2 stations away). For any

Table 3.10. HSDB Token Frame

Parameter	#Bits
Preamble	16
Start Delimiter (SD)	4
Bit 4	1
Destination Address (DA)	7
Frame Check Sequence (FCS)	8
End Delimiter (ED)	4

Table 3.11. HSDB Message Frame Overhead

Parameter	#Bits
Preamble	16
Start Delimiter (SD)	4
Frame Control (FC)	8
Source Address (SA)	8
Destination Address (DA)	16
Word Count (WC)	16
Frame Check Sequence (FCS)	16
End Delimiter (ED)	4

number of stations, n, on the ring, this number would be (n-1)/2. By not implicitly modeling token passage, the output Bus utilization represents productive utilization. With token passing, including it is always 100% busy. Using this assumption, the delay assigned to nodes named **WAIT4TOKEN** was calculated as the time to pass a token frame (and preamble) plus the intertransmission gap or

```
40 bits x 0.02 microsecond/bit + 0.28 microread = 1.08 microsecond
```

For the BIU, MAB, and BTB nodes with a message overhead of 88 bits, the calculation is

```
(N x 16 + 88) x 0.02 microsecond
```

where N is the number of words in the message. Table 3.12 provides the HSDB firing delays for various message lengths.

3.4.2. CPU and SMM Delays

The firing delays assigned to nodes mapped onto the CPU modules and onto the System Mass Memory (SMM) were estimated in three different groups:

- SUROM processing and checksums
- Message and files services

Table 3.12. HSDB TIMING

Size (words)	HSDB (microseconds)
1	2.08
2	2.40
3	2.72
4	3.04
6	3.68
8	4.32
9	4.64
17	7.20
18	7.52
20	8.16
22	8.80
23	9.12
28	10.72
29	10.93
30	11.36
32	12.00
40	14.56
45	16.16
46	16.48
55	19.36
60	20.96
288	93.92
520	168.16
616	198.88
1040	334.56
4096	1312.48

- Other

Most key parameters used for calculating these firing delays were provided by the TRW project manager. Some of them changed radically during the course of the modeling effort. Those that were originated with the modeling team were submitted to and either confirmed or corrected by the TRW project manager.

3.4.2.1. SUROM Processing and Checksum Delays

The key parameters for these calculations were:

Memory access time	187.5 nanoseconds
SUROM size	10,000 words
Processor speed	2.0 mips
Checksum algorithm	10 instructions cycles/word
Memory test	4 accesses/word
Memory size M1553B BIM Module	128K words
Memory size - other Modules	256K words
ISA test and discrete checks	equivalent to 256K memory test

Using these estimates one obtains the delays shown in Table 3.13.

The checksum delays from Table 3.13 are used in the appropriate checksum nodes throughout the model. Since the entire SUROM process up to transmission of the ready message is performed by the CPU without contention from other processes, this process is modeled in a single leaf node with a delay of 339,875 microseconds for the M1553B nodes and 435,875 microseconds for all others.

Table 3.13. Time for SUROM and Checksum Events

FIRING DELAY FOR SUROM AND CHECKSUMS

Read RAM to ROM	Assume 187.5 nanosecond transfer and 10,000 word SUROM.	1 ,875 microsec
Checksum	Assume 2.0 mips Processor & 10 instruction cycles/word.	50,000 microsec
Memory test	Assume 4 accesses/word & 256k memory	192,000 microsec
Memory test	Assume 4 accesses/word & 128k memory	96,000 microsec
Remainder (ISA test, other tests, Discretes)	Assume equivalent to memory test	192,000 microsec
Checksums	AS0 80,000 words	400,000 microsec
	AS1 80,000 words	400,000 microsec
	Interface 17,000 words	85,000 microsec
	Cockpit Interface 21,000 words	105,000 microsec
	Sensor Management 13,000 words	65,000 microsec
	Guidance 17,000 words	85,000 microsec
	Navigation 13,000 words	65,000 microsec
	LPU Attributes File 42 words	210 microsec

3.4.2.2. Message and File Services Delays

RTI was furnished timing data from the fifth AARTS Quarterly Review (QR5) [22]. These data showed the times in microseconds for execution of certain message services and the goals that had been specified for some of them. In accordance with instructions from the project manager, we used the lessor of the measured time or the midpoint between the measured time and the goal, whichever was less. These times are displayed in Table 3.14.

The numbers provided by QR5 were for calls from address states outside of address state 0. They thus include in and out processing through an interface unit and a BEX handler. Since some of the system messages and their connections originate within address state 0, an estimate was needed for the amount of time to be ascribed to their processing. In addition, estimates were needed for the files services (open, close, read, write). A matrix of procedures, functions, etc., called by the measured and unknown services was constructed. Table 3.15 contains this matrix at the top. Each row of the matrix describes a service. Each column of the matrix describes a function that induces a delay. An x is placed in the matrix if that function is required to perform the service. The total delay for a service should equal the sum of the delays for the marked functions. The left column of the key defines the column headings for the matrix. The known (estimated) times were placed in the time column and the unknown ones (marked by asterisks) were calculated using the algorithms and assumptions shown in the lower right portion of the table. The resulting times for the services provided

Table 3.14. Time for Message I/O Services

FIRST PASS FIRING DELAYS (Used min(QR5,(QR5+Goal)/2)

Operation	Goal	QR5	Use
Connect Receive	210	357	283
Redirect	75	215	145
Flush	75	190	132
Disconnect Receive	210	223	217
Connect Transmit	210	382	296
Transmit Nowait	135	474	305
Transmit	135	562	348
Disconnect Transmit	210	223	217
Event Create		193	193
Event Set	225	103	103
Event Clear	225	103	103
Event Toggle	225	120	120
Event Polarity_Of	75	104	90
Event Is_Created		101	101
Event Delete		185	185
Semaphore Create		191	191
Semaphore Acquire	105	104	104
Semaphore Release	195	116	116
Semaphore Delete		177	177
Critical Section Enter	135	96	96
Critical Section Leave	135	96	96
Calendar Seconds (Clock)		109	109
Simple Accept		321	321

within address state 0 were placed in the column headed "(AS0)." The remark at the end of the connect and disconnect rows emphasizes that a transmission of the channel array to the HSDBIM must be added to each HSDB connect and disconnect. These are the delays used in the model.

3.4.2.3. Other Delays

Delays associated with AARTS algorithms such as processing the Mission Database, computing a configuration, stopping a CPU, etc., were estimated jointly by TRW and RTI personnel. The same was done for the LPU algorithms, the Bus Interface Module Software, and for System Mass Memory processing. These estimates are displayed in Table 3.16. The firing delays have all been entered into the model and the script files for setting firing delays with a decimal point and two zeros. This will facilitate locating them for change (in the script files) should better estimates or actual measurements become available.

Table 3.15. Time for Message and Files Services

FIRING DELAYS FOR MESSAGE_IO AND FILES SERVICES

SERVICE	a	b	c1	c2	c3	d1	d2	d3	e1	e2	e3	e4	e5	f	g	h	i	time (ASO)
Connect Receive	X	X	X			X			X						X		X	284 (184) Add Trans f/HSDB
Connect transmit	X	X	X			X				X							X	296 (196) Add Trans f/HSDB
Open	X	X		X	X		X		X	X	X			X		X	X	*574 (474)*
Transmit	X	X		X							X			X			X	348 (248)
Transmit nowait	X	X		X							X						X	305 (205)
Read	X	X		X				X	X	X	X			X			X	*489 (389)*
Write	X	X		X				X	X	X	X			X			X	*489 (389)*
Flush/Redirect	X	X		X													X	132/145
Disconnect Receive	X	X		X								X			X		X	217 (117) Add Trans HSDB
Disconnect Transmit	X	X		X									X				X	217 (117) Add Trans HSDB
Close	X	X		X								X					X	*187 (87)*

KEY

a: Initial processing through I/F and BEX handler.
b: Check for null connection/channel record.

c:
- c1: Check Authorization.
- c2: Check connection/channel record for ready/open.
- c3: Check Authorization w/security.

d:
- d1: Allocate & initialize connection
- d2: Allocate & initialize channel record.
- d3: Allocate label

e:
- e1: Enable label.
- e2: Create CCB.
- e3: Execute CCB.
- e4: Disable label.
- e5: Delete CCB.

f: Wait.
g: Check for primary connector...
h: Initialize Inquiry request Msg.
i: Final processing through BEX and I/F.

CALCULATION

```
from transmit & nowait              f = 43
from flush & discon_Trans          e5 = 72
from connects                      e2 = e1 + g + 12
from disconnects                   e5 = e4 + g = 72
assume e1 ~ e4
then get                           e2 = 72 + 12 = 84
from con & dis receive c1 + d1 + e1 = c2 +e4 + 67
assume c1 ~ c2
then get                           d1 = 67
from con trans & trans nowait      e3 = d1 +e2 + 9
or                                 e3 = 160
this leaves          a + b + c1/2 + i = 145
if assume c1/2 = 30 and b = 15  a+i = 100
assume e1/4 = 42 & g = 30     Close = 217 - 30 = 187
assume d3 = 15 and get   read/write = 145 + 15 + 42
                         + 84 + 160 + 43 = 489

assume c3 = 45, d2 = 40, and h = 15 and get:
 open = 145 + 40 + 40 + 42 + 84 + 160 + 43 + 15 = 574
```

Table 3.16. Assumed Firing Delays

TRANSMIT TWO WORD (XMIT)	100.00
STARTUP.ARBCPU11 (or21).CALLCLUSTERARB	100.00
STARTUP.ARBCPU12 (or22).ARBCLUSTER.CALLCLUSTERARB	50.00
All MONITORxxHB	50.00
xSMMx.XMITSTATUS	348.00
SMM send block	1000.00
HSDBIM process msg from PI to HSD Bus	200.00
HSDBIM process msg from HSDB	20.00
STARTUP.ARBCPU22.ASSUMESYSSUP.STARTSYSMGMT	400.00
STARTUP.ARBCPU22.ASSUMESYSSUP.STOPHOTHB	50.00
CPU Process Mission Database	200.00
Compute Configuration/Reconfiguration	600.00
MAB UPDATE ARRAY	200.00
All RCV system msg in CPU	50.00
DGS CALCULATE	300.00
CHANGE WAYPOINT	200.00
COMPUTE STEER	800.00
COMPUTE AIRDATA	600.00
COMPUTE INSDATA	1800.00
COMPUTE MMP	150.00
COMPUTE IMFK	400.00
COMPUTE WAYPOINT	500.00
COMPUTE shutdown, STOP CPU, STOP LPU	100.00
UNLOAD LPU or AS1	5000.00
UPDATE Structures	50.00

3.5. Flow Control

A GIPSIM simulation is controlled by the passage of tokens between nodes. This control is effected by the assignment of values to the attributes *token_consume_rate*, *token_produce_rate*, *firing_threshold*, and *initial_token_count* for node in and outports and to the attribute *queue_size* for arcs. In most cases, these attributes are set to the default value 0 for *initial_token_count* and 1 for the others. The following paragraphs will describe how these attributes are manipulated to control the execution of the model.

Figure A-8 is the graph of node **SUROMBTB2** on the startup graph. Values assigned for the node in and outports in this graph are shown in Table 3.17.

Table 3.17. Port Attributes - Graph of Node SUROMBTB2

Node	Port	Consume	Threshold	Initial	Produce
WAIT4TOKEN	0	1	1	1	*inport*
RUNCHECKSUM	0	20	20	0	*inport*
RUNCHECKSUM	0	*outport*	*n/a*	*outport*	0
join2	0	2	2	0	*inport*
split1	0	25	25	24	*inport*
split2	0	1	1	20	*inport*
split2	1	1	2		*inport*
split2	0	*outport*	*outport*	*n/a*	0
SETUPBTBDOWNLOAD	0	*outport*	*outport*	*n/a*	2
all others	all	1	1	0	*inport*
all others	all	*outport*	*outport*	*n/a*	1

The graph commences execution when a token is received at expiration of the power-up delay (through node **inport0**) on the inport of node **STARTSUROM**. After node **START-**

SUROM's firing delay has passed, a token is passed to node SETUPBTBDOWNLOAD (inport 0). SETUPBTBDOWNLOAD is an "OR" node and so will execute upon receipt of this token. Two tokens produced by outport 0 of node SETUPBTBDOWNLOAD. These provide for two primes of *inport1* on node WAIT4TOKEN (consume and threshold both 1). The single initial token on *inport0* of node WAIT4TOKEN allows the node to execute upon receipt of the tokens from node SETUPBTBDOWNLOAD. This use of an initial token is the standard way to initialize cycles. After the first transmission of the 2-word message, node delay waits for the assigned time and then outputs a token to inport 0 of node WAIT4TOKEN providing for a second execution of the message cycle. When node delay executes the second time, both tokens have been consumed on *inport1* of node WAIT4TOKEN and the cycle of 2-word messages terminates. Node join2 connects to graph outport 1 that represents the arc from node SUROMBTB1 to node SMMAS0TOBTB on the parent graph. Node join2 is an OR node; that is it will execute whenever the token threshold is met on any, rather than all, of its inport queues. The *firing_threshold* and *token_consume_rate* on *inport0* of node join2 has been set for execution after two message transmissions. Node split1 activates the receipt of the 4-word message. It must execute upon receipt of the first token from the SMM and not on any of the following. This is achieved by setting the *token_consume_rate* and *firing_threshold* to a number (in this case 25) greater than the total number of expected incoming tokens (in this case 24) and setting the *initial_token_count* to one less. The queue size on the arc must be set 25 or more or a full queue will block the predecessor node. Node split2 activates the receipt of a 4K block of data from the SMM during

download of the AS0. The feedback arc from **outport0** (zero *token_produce_rate*) to **inport0** (a *token_consume_rate* and *firing_threshold* of 1 and *initial_token_count* of 20) ensures that this node will execute no more than 20 times. Since the sequence of 4K blocks follows the receipt of the 4-word message, the *firing_threshold* on *inport1* has been set to 2 with a *token_consume_rate* of 1 and no *initial_token_count*. This causes node **split2** to commence execution upon receipt of the second token from the memory and to continue to execute on each succeeding token until the 20 tokens initially on *inport0* are consumed. Again the queue sizes must accommodate the maximum number of tokens. Node **RUNCHECKSUM** is set to execute after receipt of twenty 4K blocks of data from the SMM. *outport1* of node **RUNCHECKSUM** would, if it produced a token, restart the entire process. Since the guidelines for the simulation was to assume no failure of BIT or checksum, the *token_produce_rate* has been set to 0.

Node **READLPULOC** is an internal node. Therefore, the execution takes place in its subgraph (Figure A-9). The *token_consume_rate*, *firing_threshold*, and *token_produce_rate* attribute values are shown in Table 3.18.

Table 3.18. Port Attributes - Graph of Node READLPULOC

Node	Port	Consume	Threshold	Initial	Produce
RCVRESPONSE	0	3	3	2	*inport*
RCVDATA	0	3	3	1	*inport*
RCVSTATUS	0	3	3	0	*inport*
split	0	1	22	0	*inport*
all others	all	1	1	0	*inport*
all others	all	*outport*	*n/a*	*outport*	1

In this case, we want nothing to happen until after the 4-word message and AS0 have been received (21 messages). Thus, we put a threshold of 22 on node split which receives the SMM inputs. The consume of 1 assures that it will execute on each received message after the 21st. Three messages will be received from the SMM: the open file response, the data, and the read status. By putting a threshold and consume of 3 on the three corresponding nodes and initial token counts of 2, 1, and 0, we achieve the proper sequence of execution.

Figure A-10 is the graph of node **SMMAS0TOBTB** on the startup graph. The *token_consume_rate*, *firing_threshold*, and *token_produce_rate* attribute values are shown in Table 3.19.

Table 3.19. Port Attributes - Graph of Node SMMAS0TOBTB

Node	Port	Consume	Threshold	Initial	Produce
WAIT4TOKEN0	0	3	3	2	*inport*
WAIT4TOKEN0	1	20	20	0	*inport*
WAIT4TOKEN1	0	1	1	1	*inport*
WAIT4TOKEN2	0	0	0	0	*inport*
WAIT4TOKEN2	1	3	3	2	*inport*
WAIT4TOKEN3	0	1	1	1	*inport*
XMITDATA0	0	*outport*	*outport*	*n/a*	20
XMITDATA1	0	*outport*	*outport*	*n/a*	20
split1	2	*outport*	*outport*	*n/a*	0
all others	all	1	1	0	*inport*
all others	all	*outport*	*outport*	*n/a*	1

In this graph we want the nodes **WAIT4TOKEN0** and **WAIT4TOKEN2** to execute on receipt of the first message from the appropriate BTBIM with the 4-word message followed by the 20 4K blocks of AS0. The next two messages are the open and

read for the LPU attributes file. They will trigger events in the subgraph of node **RDLPULOCx**. Thus, we place a threshold and consume of three with an initial count of two on inports 0 and 1 of nodes **WAIT4TOKEN0** and **WAIT4TOKEN2**, respectively. The consume and threshold of 0 on inport 0 of node **WAIT4TOKEN2** allows the node to execute on inputs to inport 1 only. The arc from node **split3** to node **WAIT4TOKEN2** provides a means to allow simultaneous loading of the BTBIMs by putting a produce, consume, and threshold of 1 on the parts associated with this arc and the arc from node **split3** to node **WAIT4TOKEN0** and an initial token in the inport of the node we want to "fire." First this graph can be made to transmit the 4K blocks alternately to the BTBIMs. As currently configured, the consume and threshold of 20 on inport 1 of node **WAIT4TOKEN0** prevents it from executing until the BTB2 load is complete.

Nodes **RDLPULOCx** are internal nodes with graphs as shown in Figure A-11. The *token_consume_rate*, *firing_threshold*, and *token_produce_rate* attribute values are shown in Table 3.20.

Table 3.20. Port Attributes - Graph of Node RDLPULOCx

Node	Port	Consume	Threshold	Initial	Produce
RCVOPENLPULOC	0	2	2	1	*inport*
RCVRDLPULOC	0	2	2	0	*inport*
split	0	1	2	0	*inport*
all other	all	1	1	0	*inport*
all other	all	*outport*	*outport*	n/a	1

For this graph we have two trains of events, each initiated by an incoming token. Execution must start with the second token received from the BTBIM (see discussion

of Figures A-8 and A-10). The threshold of 2 and consume of 1 on node **split** with no initial token assures that it will execute on receipt of the second incoming token and any subsequent ones. The threshold and consume of 2 with 1 initial token on node **RCVOPENLPULOC** means it will execute on receipt of the first token from node **split** (the open request) and not on the second. The threshold and consume of 2 with no initial token on node **RCVRDLPULOC** means it will execute on the second token received from node **split**.

Figure A-15 is the graph of the BTBIM receiving messages from a module and passing them to the recipient during the loading of the bootstrap loader and AS0 software. Table 3.21 shows the port attributes assigned to nodes in this graph.

Table 3.21. Port Attributes - Graph of Node BTBTOSMMAS0

Node	Port	Consume	Threshold	Initial	Produce
RCV2WORD	0	26	26	25	inport
RCVREADREQ	0	1	2	0	inport
RCVRESPONSE	0	50	50	49	inport
All Other	-	1	1	0	inport
All Other	-	outport	outport	n/a	1

We want node **RCV2WORD** to execute on the first token received from the module and then to ignore any others. We thus set a consume and threshold greater than the expected number of incoming tokens (26) and assigned one less to the initial token attribute. Node **RCVREADREQ** must pass all subsequent tokens to the SMM. Therefore, we put on it a consume of 1 and a threshold of 2 with no initial token. Finally, when the memory responds to the open request for the bootstrap loader (this

request is initiated in this graph) this graph transmits a read request to the SMM. We want this chain to execute on the response to the open file response and not to any subsequent SMM responses. We achieve this with a threshold and consume of 50 and an initial token count of 49.

Figure A-16 is the graph of the BTBIM passing messages from the memory (and in one case from the graph discussed in the preceding paragraph) to a module. Table 3.22 shows the port attributes assigned to nodes in this graph.

Table 3.22. Port Attributes - Graph of Node BTBTOCPUAS0

Node	Port	Consume	Threshold	Initial	Produce
Split	0	1	2	0	inport
RCVATTRIBRESP	0	2	44	0	inport
RCVFILERESP	0	2	3	0	inport
RCVFILERESP	1	1	1	21	inport
RCVBOOTRESP	0	50	50	48	inport
RCVBOOT	0	50	50	49	inport
RCVAS0	0	2	4	0	inport
RCVAS0	1	1	1	20	inport
RCVATTRIB	0	45	45	0	inport
RCVFILERESP	1	outport	outport	n/a	0
RCVAS0	1	outport	outport	n/a	0
All Other	-	1	1	0	inport
All Other	-	outport	outport	n/a	1

The first token passed from the SMM hierarchy into this (BTBIM) hierarchy is the response to the bootstrap loader open request. This triggers a read request in the graph discussed in the preceding paragraph. The token consume of 1 and threshold of 2 on node split in this graph assures that the first token from SMM will be ignored in this graph while each succeeding token will cause an output. Commencing with

the second incoming token, node **split** places a single token on each of its outport queues each time a token is received from memory. The first of these tokens represents transmission of the bootstrap loader. This token causes node **RCVBoot** to execute. The consume of threshold of 50 with an initial count of 49 causes this node to execute on the first token received (and, were there to be that many, on the 51st). This is followed by a read status message from the SMM to the BTBIM. Node **RCVBOOTRESP**, with a consume and threshold of 50 and an initial token count of 48, executes on the second token from node **split**. After the bootstrap loader has been received by the module, the module transmits an open file request for the AS0 software. This causes transmission of a token representing the open file response from the SMM. Node **RCVFILERESP** has two inports and two outports. Inport 0's execution condition will be satisfied upon receipt of the third token from node **split** and every second token thereafter. Inport 1 is assigned a consume and threshold of 1 with 21 initial tokens while outport 1 has a produce of zero. This means that inport 1's conditions will be satisfied 21 times and then it will block further execution of the node. Node **RCVAS0** is similarly configured to receive 20 blocks of the AS0 software commencing with the fourth token from node **split**. Thus, nodes **RCVFILERESP** and **RCVAS0** alternate during the simulated download of the 80K AS0 file (open response followed by 20 blocks of data each followed by a read status). Finally, nodes **RCVATTRIBRESP** and **RCVATTRIB** execute in a similar manner on the 44th through 46th (final) token received from node **split**.

Other types of employment of port attributes for flow control can be seen in the arbi-

tration graphs. Figure A-12 is a graph of a module receiving its bootstrap loader and AS0 software. For CPU 22 (and 12), the final node to execute, node CHECKSUMATTRIB places 5 tokens on the outgoing arc (node outport1) that connects nodes AS0CPU22 and ARBCPU22 on the startup graph. Figure A-58 is a subgraph of node ARBCPU22 where the tokens are received by inport 0 of node CALLCLUSTERARB. Inport 0 and inport 1 of node CALLCLUSTERARB have a consume and threshold of 1 assigned. In addition, inport 1 has an initial token to initiate the cycle through the delay node on the parent graph (Figure A-19). This provides for five executions of this subgraph. Each execution places a token on the arc connecting node ARBCLUSTER with node LOADAS1 on the parent graph (Figure A-19). The receiving node in the graph of node LOADAS1 (Figure A-59) is node OPENAS1. The inport of this node is assigned a consume and threshold of 5 assuring that the graph will execute once after five cycles through the ARBCLUSTER graph. Port attributes, similar to those assigned for the loading of the AS0 software, control the alternation between receiving responses (and transmitting reads) and receiving data in the right-hand portion of this graph. After execution, node RUNCHECKSUM places a token on the arc connecting node LOADAS1 to node ARBHB on the parent graph. This token causes a single execution of node MONITORHB in the subgraph of node ARBHB, (Figure A-60). Node MONITORHB sends a token to node OR0, which executes upon receipt of a token on any of its inports. Node OR0 produces a token that initiates the subgraph of node TRANSMITHB. This subgraph places a token on the outport connecting to node delay, which initiates subsequent executions of the cycle, and on a graph port toASSUMESSYSSUP, which connects

node **ARBHB** to node **ASSUMESYSSUP** on the parent graph. These tokens enter the graph of node **ASSUMESYSSUP**, Figure A-61, where they enter the subgraph of node **ARBSYSSUP**, Figure A-62, through graph port **start** and into node **MONITORSSHB**. Inport 0 of node **MONITORSSHB** has been assigned a threshold and consume of 7 with an initial token count of 2. This causes the graph to initiate upon receipt of the fifth token from the hot backup heartbeat, and not to initiate unless it receives seven more. The module must issue five non-colliding system supervisor heartbeats before assuming the role of system supervisor and terminating the hot backup heartbeat. The number 7 allows for six hot backup heartbeats to occur without reinitiating the graph during this period. As it turns out, only 5 occur. Any larger number could have been used. Upon execution, node **MONITORSSHB** places five tokens on the arc to node **TRANSMITSSHB** which provides for five executions of the subgraph of that node (there is an initial token on the arc from node **delay**). Each execution of this subgraph, in addition to initializing the delay, places a token on each of the graph outports. These tokens are passed to nodes **STARTSYSMGT** and **STOPHOTHB** on the parent graph, Figure A-61. The *inport* for each of these nodes is assigned a consume and threshold of 5; so, they will execute after five heartbeats. The four graph outports connect to the system messages function where the supervisor role is initiated (including continuation of the heartbeat); to node **LPUCPU22** on the startup graph to allow starting of LPU loading (the outport is misnamed "tonormalops"); to the **ARBHB** node on the parent graph to stop the hot backup heartbeat; and to CPU12 to reinitiate arbitration for hot backup. Each of these nodes passes a single token

except the one to **ARBHB** which passes 40 (any large number would do). Referring to Figure A-60, the 40 incoming tokens through graph inport **stop** cause node **OR0** to fire repeatedly (firing delay of zero). This results in tokens simultaneously on all arcs of the cycle through node **delay**. Since the queue size is 1 for each of these arcs, none of the nodes can execute. There is no place for an output token. This stops the cycle. The same approach is used to stop heartbeats and LPUs on simulated failure and on shutdown.

In the discussion of the AS0 load, we encountered the feedback arc from an outport to an inport of the same node with a 0 token produce and initial tokens sufficient to provide the required number (and no more) of executions. This method is employed on the top-level graph (Figure A-7) to restrict nodes **POWERUP** and **FAILURE** to a single execution. The loop on node **POWERUP** appears redundant given those on the nodes **pwrondlyx** on the startup graph (Figure A-7). When the entire model is running, either approach would be sufficient. However, if one wants to run the startup graph as a model itself, the loops on that graph are required. If one wants to delete the processing time associated with simulating startup in order to investigate or debug later portions of the model, he can change the attribute *node_class* of node **STARTUP** on the top-level graph to leaf. Then the loop on node **POWERUP** is required. A similar loop can be found on node **COMPUTENEWCONFIG** in the reconfigure graph (Figure A-49) and on node **CPU22RcvShutDn** in the shutdown graph (Figure A-52). These allow running these portions of the model independently.

4. Results

In this section we discuss model validation and provide some simulation outputs that show how the ADAS model can be used to assess timing, performance, resource utilization and the sensitivity of these measures to input parameters.

The original purpose of this model was to calibrate the abstraction against measurements of the AARTS demonstration 3 prior to expansion of the model for analysis of a full PAVE PILLAR mission applications system. The AARTS demonstration 3 has been delayed so calibration has not been conducted. The representation of the modeled AARTS and LPU functions has been validated through continuous interchange of model and AARTS design concepts throughout the project. In fact, much of the effort in constructing the model was devoted to model changes reflecting design changes in the ongoing AARTS development. Many of these design changes were initiated by the discovery, in the modeling effort, of future problems should the design continue in its current direction at that time. While the functionality of the model is verified, most of the numbers assigned for resource utilization still rest upon tenuous estimates. Before expanding the model, these estimates should be replaced by better estimates or by measured values. Following this, performance outputs such as those in the following tables can then be compared with actual performance and any necessary changes made to the model architecture.

In the remainder of this section, several tables are displayed and discussed. These ta-

bles show resource utilization and timing of events in simulations run with the model. They are discussed in the context of how design characteristics input parameters or modeling assumptions impact the various processes. All of the results are extracted from ADAS outputs. Table 4.1 shows the commencement and completion times of major stages of the basic model.

Table 4.1. ADAS Model: Subprocess Timing

Sub-Process Name	Simulation time Commencement (μs)	Simulation time Completion (μs)	Comments
Startup		5177719	Up through loading of LPUs
Normalops_1 OPS	5174636	6650000	Until CPU11 fails
Reconfiguration	6650000	8017927	Reallocation of LPUs
Normalops_2	8017927	9183419	CPU11 now off-line
Shutdown	9183419	9296302	CPU22 last to shut down

Table 4.2 displays the completion time of key events during system startup. This (table) focuses upon the loading of the AS0 software into modules, the loading of AS1 software into supervisory modules, and the loading of LPUs into the CPUs. These activities terminate with the completion of the checksum on the download. The current model configuration requires completion of the download of the bootstrap loader, the AS0 software (including checksum), and the LPU attributes file into one module before loading of the next module commences. The AS1 software is loaded into supervisory modules when they are ready without such a restriction. The only constraint in the loading of AS1 software is contention with ongoing AS0 loads for shared resources. LPU loading (including the mission database file into supervisory

Table 4.2. Startup with Assigned Delays

TIME (microsec)	HARDWARE MODULE	EVENT
1449514	MAB2CPU	STARTUP:AS0MAB2:CHECKSUMAS0
1936773	MAB1CPU	STARTUP:AS0MAB1:CHECKSUMAS0
2424031	CPU22CPU	STARTUP:AS0CPU22:CHECKSUMAS0
2911896	CPU12CPU	STARTUP:AS0CPU12:CHECKSUMAS0
3419078	CPU21CPU	STARTUP:AS0CPU21:CHECKSUMAS0
3449520	CPU22CPU	STARTUP:ARBCPU22:LOADAS1:RUNCHECKSUM
3931165	CPU11CPU	STARTUP:AS0CPU11:CHECKSUMAS0
3948036	CPU12CPU	STARTUP:ARBCPU12:LOADAS1:RUNCHECKSUM
4418746	M1553B2CPU	STARTUP:AS01553B2:CHECKSUMAS0
4908187	M1553B1CPU	STARTUP:AS01553B1:CHECKSUMAS0
5017493	CPU11CPU	STARTUP:LPUCPU11:readlpu1:RUNCHECKSUM
5039873	CPU21CPU	STARTUP:LPUCPU21:readlpu1:RUNCHECKSUM
5097436	CPU11CPU	STARTUP:LPUCPU11:readlpu2:RUNCHECKSUM
5142472	CPU21CPU	STARTUP:LPUCPU21:readlpu2:RUNCHECKSUM
5176190	CPU11CPU	STARTUP:LPUCPU11:readlpu3:RUNCHECKSUM
5177719	na	STARTUP:ENDSTARTUP

modules) commences only after all modules have completed the AS0 download. The LPUs are loaded sequentially in an individual CPU. Separate CPUs contend for resources during LPU loading. It can be seen from Table 4.2 that the AS0 download for a module takes 0.49 seconds (rounded), with those conducted concurrent with AS1 downloads (CPU21 and CPU11) taking 0.51 seconds due to resource contention. The delay for checksum of the 80k file is 0.4 seconds and the bus transfer time for 82k words is 0.036 seconds. This means that 0.054 seconds of delay are attributed to the functioning of the BTBIM, SMM, CPU (open and read command, etc.), and read status reporting. The remainder of the table shows the interleaving of the LPU reads.

Table 4.3. Failure and Reconfiguration with Assigned Delays

TIME (microsec)	HARDWARE MODULE	EVENT
6650000	failure	FAILURE
7900000	detect	DETECT:DETECTMISNGPLSE
7901450	CPU22CPU	RECONFIGURE:COMPUTNEWCONFIG
7997302	CPU12CPU	CP12LOADLPU:CPU12LOADLPU:RUNCHECKSUM
7997679	CPU22CPU	CP22LOADLPU:CP22LOADLPU:RUNCHECKSUM
8017927	CPU21CPU	CP21LOADLPU:CPU21LOADLPU:RUNCHECKSUM

Table 4.3 shows similar data for the period from the simulated failure of CPU11 through the reloading of the LPUs into other CPU modules. As can be seen, the time to react to a failure is dominated by the time to detect the failure (10 missed pulses at 8Hz = 1.25 seconds). From detection of the failure through reloading and checksum of all three LPUs took 0.12 seconds.

Table 4.4 shows the sequence of events during shutdown. The events, STOPx, are the final stopping of the indicated module. The shutdown process took just over 0.11 seconds. Also shown are the time of completion of each LPU unload. The model, as currently configured, assumes that all LPUs are stopped before unloading commences.

Table 4.5 shows resource utilization during selected time periods. The utilization of bus interface units and the MI553B module was much smaller than that of the modules displayed in Table 4.5 and is not shown in order to reduce the size of the table. This table shows that startup through arbitration utilizes the specialist and MAB CPUs 19%. The BTBCPUs have a slightly higher utilization as they broker the downloads,

Table 4.4. Shutdown with Assigned Delays

TIME (microsec)	HARDWARE MODULE	EVENT
9183419		Token received to start shutdown
9184235	M1553B2CPU	Cluster2ShutDn:STOP0
9184237	BTB2CPU	Cluster2ShutDn:STOP1
9215986	CPU21CPU	Cluster2ShutDn:CPU21UnloadLPU:UnloadLPU1
9221086	CPU21CPU	Cluster2ShutDn:CPU21UnloadLPU:UnloadLPU2
9226086	CPU21CPU	Cluster2ShutDn:CPU21UnloadLPU:UnloadLPU3
9226700	CPU21CPU	Cluster2ShutDn:STOP2
9228205	M1553B1CPU	Cluster1ShutDn:STOP0
9228207	BTB1CPU	Cluster1ShutDn:STOP1
9255165	CPU12CPU	Cluster1ShutDn:CPU12UnloadLPU:UnloadLPU1
9260165	CPU12CPU	Cluster1ShutDn:CPU12UnloadLPU:UnloadAS1
9260979	CPU12CPU	Cluster1ShutDn:STOP2
9261865	MAB1CPU	Cluster1ShutDn:STOP3
9262532	MAB2CPU	Cluster2ShutDn:STOP3
9291252	CPU22CPU	Cluster2ShutDn:CPU22UnloadLPU:UnloadLPU1
9296252	CPU22CPU	Cluster2ShutDn:CPU22UnloadLPU:UnloadAS1
9296302	CPU22CPU	Cluster2ShutDn:STOP4

Table 4.5. Resource Utilization (Standard Delays)

	Start Thru Arbitration	LPU Loading Initial	LPU Loading Reconfigure
Time Increment	4.451177	0.723211	0.115461
BTB1CPU	21%	3%	3%
BTB2CPU	21%	1%	6%
CPU11CPU	19%	31%	0%
CPU12CPU	28%	2%	64%
CPU21CPU	19%	27%	78%
CPU22CPU	28%	3%	62%
MAB1CPU	19%	1%	2%
MAB2CPU	19%	2%	2%
BTB	7%	7%	12%
MAB	0%	0%	0%
PIBUS1	1%	1%	1%
PIBUS2	1%	1%	3%
SMM	11%	11%	16%

The supervisory modules, CPU12 and CPU22, show the highest CPU utilization since they must download the AS1 software. The utilization of the buses is small. During the LPU loading, CPU11 (3 LPUs) and CPU21 (2 LPUs) are significantly busy. The low utilization of the BTBCPUs and the BTB reflect the fact that as many as 3 LPUs are serially loaded into a single CPU with a large checksum delay between them. The final column shows the utilization during the reconfiguration. LPU21 has two LPUs continuing to operate during this period. A higher BTBCPU and BTB utilization is achieved since loads and checksums can be achieved in parallel (1 LPU was loaded into each of the three remaining CPU modules).

As stated before, the only "measured" data specific to AARTS that was available was

the simulation results for message IO functions presented in quarterly review number 5 (QR5). For the model, we used the mid-point between these numbers and the goals specified for these services. We ran a simulation using the actual QR5 numbers for comparison. The method for estimating numbers for unknown message IO and files services used for this effort was the same as used for the basic model. Table 4.6, a duplicate of Table 3.15 except for the numbers, shows the new calculations and results. Tables 4.7, 4.8, and 4.9 show the timing with these numbers for the same events as those shown in Tables 4.2, 4.3, and 4.4. We see that a large increase in the CPU times required for files services had only a small effect on the startup times, about a 3% increase. Startup time is dominated by the checksums. The effect during reconfiguration was about twice as large (6%), while that during shutdown (mostly message oriented) was 8%. Table 4.10 shows resource utilization using the QR5 delays. This table can be compared with Table 4.5. As can be seen, the larger messages and files services times had little effect on resource utilization.

Finally, runs were conducted using the two sets of firing delays during a period of normal operations. Each period commenced with the completion of the startup and configuration processes. Each was run for an identical simulated time period. All LPU outputs fired the same number of times during the time period for each set of firing delays. Table 4.11 shows the utilization of the CPUs during this time period. The impact of the increased message passing time, while small, is more pronounced during normal operations than during the file loading periods where checksums dominate.

Table 4.6. Time for Message and Files Services (QR5 Numbers)

FIRING DELAYS FOR MESSAGE_IO AND FILES SERVICES
(using QR5 numbers)

SERVICE	a	b	c1	c2	c3	d1	d2	d3	e1	e2	e3	e4	e5	f	g	h	i	time (ASO)
Connect Receive	X	X	X			X			X						X		X	357 (247) Add Trans f/HSDB
Connect transmit	X	X	X			X				X							X	382 (272) Add Trans f/HSDB
Open	X	X		X	X		X		X	X	X			X		X	X	*792 (682)*
Transmit	X	X		X							X			X			X	562 (452)
Transmit nowait	X	X		X							X						X	474 (364)
Read	X	X		X				X	X	X	X			X			X	*667 (557)*
Write	X	X		X				X	X	X	X			X			X	*667 (557)*
Flush/Redirect	X	X		X													X	190/215
Disconnect Receive	X	X		X								X			X		X	223 (113) Add Trans HSDB
Disconnect Transmit	X	X		X									X				X	223 (113) Add Trans HSDB
Close	X	X		X								X					X	*205 (95)*

KEY

a: Initial processing through I/F and BEX handler.
b: Check for null connection/channel record.

c: c1: Check Authorization.
c2: Check connection/channel record for ready/open.
c3: Check Authorization w/security.

d: d1: Allocate & initialize connection
d2: Allocate & initialize channel record.
d3: Allocate label

e: e1: Enable label.
e2: Create CCB.
e3: Execute CCB.
e4: Disable label.
e5: Delete CCB.

f: Wait.
g: Check for primary connector...
h: Initialize Inquiry request Msg.
i: Final processing through BEX and I/F.

CALCULATION

from transmit & nowait f = 88
from flush & discon_Trans e5 = 33
from connects e2 = e1 + g + 25
from disconnects e5 = e4 + g = 33
assume e1 ~ e4
then get e2 = 33 + 25 = 58
from con & dis receive c1 + d1 + e1 = c2 +e4 + 134
assume c1 ~ c2
then get d1 = 134
from con trans & trans nowait e3 = d1 +e2 + 117
or e3 = 276
this leaves a + b + c1/2 + i = 200
if assume c1/2 = 60 and b = 30 a+i = 110
assume e1/4 = 15 & g = 18 Close = 223 - 18 = 205
assume d3 = 30 and get read/write = 200 + 30 + 15 + 58 + 276 + 88 = 667

assume c3 = 75, d2 = 80, and h = 30 and get:
open = 200 + 75 + 80 + 15 + 58 + 276 + 88 + 30 = 792

Table 4.7. Startup with QR5 Delays

TIME (microsec)	HARDWARE MODULE	EVENT
1466686	MAB2CPU	STARTUP:AS0MAB2:CHECKSUMAS0
1972272	MAB1CPU	STARTUP:AS0MAB1:CHECKSUMAS0
2477519	CPU22CPU	STARTUP:AS0CPU22:CHECKSUMAS0
2982934	CPU12CPU	STARTUP:AS0CPU12:CHECKSUMAS0
3509585	CPU21CPU	STARTUP:AS0CPU21:CHECKSUMAS0
3532701	CPU22CPU	STARTUP:ARBCPU22:LOADAS1:RUNCHECKSUM
4034098	CPU11CPU	STARTUP:AS0CPU11:CHECKSUMAS0
4036766	CPU12CPU	STARTUP:ARBCPU12:LOADAS1:RUNCHECKSUM
4539370	M1553B2CPU	STARTUP:AS01553B2:CHECKSUMAS0
5045518	M1553B1CPU	STARTUP:AS01553B1:CHECKSUMAS0
5167280	CPU11CPU	STARTUP:LPUCPU11:readlpu1:RUNCHECKSUM
5185394	CPU21CPU	STARTUP:LPUCPU21:readlpu1:RUNCHECKSUM
5249141	CPU11CPU	STARTUP:LPUCPU11:readlpu2:RUNCHECKSUM
5291346	CPU21CPU	STARTUP:LPUCPU21:readlpu2:RUNCHECKSUM
5330662	CPU11CPU	STARTUP:LPUCPU11:readlpu3:RUNCHECKSUM
5332659	na	STARTUP:ENDSTARTUP

Table 4.8. Failure and Reconfiguration with QR5 Delays

TIME (microsec)	HARDWARE MODULE	EVENT
6650000	failure	FAILURE
7900000	detect	DETECT:DETECTMISNGPLSE
7901450	CPU22CPU	RECONFIGURE:COMPUTNEWCONFIG
7993796	CPU12CPU	CP12LOADLPU:CPU12LOADLPU:RUNCHECKSUM
7999823	CPU22CPU	CP22LOADLPU:CP22LOADLPU:RUNCHECKSUM
8023243	CPU21CPU	CP21LOADLPU:CPU21LOADLPU:RUNCHECKSUM

Table 4.9. Shutdown with QR5 Delays

TIME (microsec)	HARDWARE MODULE	EVENT
9339641		Token received to start shutdown
9340458	M1553B2CPU	Cluster2ShutDn:STOP0
9340460	BTB2CPU	Cluster2ShutDn:STOP1
9374756	CPU21CPU	Cluster2ShutDn:CPU21UnloadLPU:UnloadLPU1
9379756	CPU21CPU	Cluster2ShutDn:CPU21UnloadLPU:UnloadLPU2
9384756	CPU21CPU	Cluster2ShutDn:CPU21UnloadLPU:UnloadLPU3
9385721	CPU21CPU	Cluster2ShutDn:STOP2
9387507	M1553B1CPU	Cluster1ShutDn:STOP0
9387509	BTB1CPU	Cluster1ShutDn:STOP1
9414417	CPU12CPU	Cluster1ShutDn:CPU12UnloadLPU:UnloadLPU1
9419417	CPU12CPU	Cluster1ShutDn:CPU12UnloadLPU:UnloadAS1
9420081	CPU12CPU	Cluster1ShutDn:STOP2
9420968	MAB1CPU	Cluster1ShutDn:STOP3
9421635	MAB2CPU	Cluster2ShutDn:STOP3
9450354	CPU22CPU	Cluster2ShutDn:CPU22UnloadLPU:UnloadLPU1
9455354	CPU22CPU	Cluster2ShutDn:CPU22UnloadLPU:UnloadAS1
9455404	CPU22CPU	Cluster2ShutDn:STOP4

Table 4.10. Resource Utilization (QR5 Delays)

	Start Thru Arbitration	LPU Loading Initial	LPU Loading Reconfigure
Time Increment	4.509041	0.792022	0.116594
BTB1CPU	21%	4%	5%
BTB2CPU	21%	2%	7%
CPU11CPU	19%	30%	0%
CPU12CPU	28%	3%	66%
CPU21CPU	19%	27%	79%
CPU22CPU	28%	4%	63%
MAB1CPU	19%	1%	2%
MAB2CPU	19%	2%	2%
BTB	6%	7%	12%
MAB	0%	0%	0%
PIBUS1	1%	1%	1%
PIBUS2	1%	1%	3%
SMM	11%	11%	16%

Table 4.11. Resource Utilization During Normal Operations

Module	Standard Delay	QR5 Delay
CPU11CPU (3 LPUs)	16%	19%
CPU12CPU (Hot Backup)	1%	2%
CPU21CPU (2 LPUs)	9%	12%
CPU22CPU (Sys Supervisor)	2%	2%

5. Model Modification or Expansion

The details presented in Sections 3.3, 3.4, and 3.5 are intended to provide the user with sufficient understanding of the model and modeling assumptions to be able to modify the model to accommodate different assumptions or expand it for a broader scenario. Some modifications might include any or all of:

- Expansion to more or larger clusters
- Expansion to a larger number of LPUs
- Larger and/or more complex LPUs
- Different file access routes, i.e., memory modules in clusters
- Different failure patterns

This section presents further elaboration on model expansion and modification.

Figure A-63 is a top-level graph for a 4 cluster configuration. No change has been made in the LPU, failure and reconfigure portions from the graph in Figure A-6. The only changes on this graph are an increased number of arcs from nodes **STARTUP** and **SHUTDOWN** to node **SYSTEMESSAGES** and a reduction in the number of arcs from node **POWERUP** to node **STARTUP**.

Earlier, in the discussion of the startup graph, (Figure A-7) we stated that the graph contained more detail than is normal. For this 4-cluster model we incorporate the same functionality into a hierarchical expansion. Figure A-64 is a first-level startup

graph for the 4-cluster model. As can be seen, this graph shows each cluster receiving a "powerup" input and outputting the tokens to initiate the system messages for the cluster modules. The node ENDSTARTUP, as in the basic model, receives a token from each cluster indicating completion of configuration and outputs a token to initiate normal operations. This node has 0 delay and consumes no resources. In addition, each cluster not containing the system supervisor transmits a token to the system supervisor cluster, node CLUSTER3, to signify completion of arbitration and ready for configuration.

Figure A-65 shows the subgraph of node CLUSTER3. Here we see the 4 primary phases of startup represented separately. Nodes SUROM, LOADAS0, ARBITRATION, and CONFIGURE represent the cluster functioning during the respective phases. Each "phase" node is connected to a node representing the memory functions during that phase (nodes LOADAS0TOBTB, LOADAS0TOMODULES, LOADAS1TOCPU, LOADLPUS). Each of the memory function nodes would contain one-half of the hierarchy below the equivalent memory function node on the basic startup graph. For instance, node LOADAS0TOMODULES would have a subgraph that consists of one row of internal nodes and their inport and outport nodes from Figure A-17. The subgraph for the internal nodes of these "half graphs" would be the same graph used in the basic model.

For the current PAVE PILLAR concept, all memory functions are mapped onto a central system mass memory. For a concept with a memory module in each cluster, the memory function nodes would be mapped onto the separate memory module

in the cluster and the arcs between cluster functions and memory functions would represent PI-bus instead of HSDB transfers. In this case, if the cluster memory is volatile, we would need a graph inport to a memory node coming from the SMM (which would be represented on the parent or a higher level graph) for loading the cluster modules. Looking again at Figure A-65 as a whole, we can see that the BTB actively loads the AS0 software in the **SUROM** node and outputs a token to initiate its system messages. All modules output tokens to initiate the AS0 passive loads. Passive loading, node LOADAS0, outputs tokens as modules start the AS0 software to initiate the appropriate system messages. In addition, the CPUs output initiation tokens to the arbitration process and the MAB signifies "ready" for configuration. Upon completion of arbitration, the system supervisor (or Hotbackup or cluster supervisor) outputs a token to initiate supervisory messages and the CPUs signify "ready" for configuration. In the other clusters, the cluster supervisor signifies "ready" to cluster 3. These are the three external inputs to node **CONFIGURE**. When the configuration is completed a token is output to node **ENDSTARTUP** on the parent graph.

Two nodes on Figure A-65 have been expanded to show the connection to the graphs in the basic model. Figure A-66 is the subgraph of node **SUROM** on Figure A-65. It is readily seen that this graph is the upper left corner of the original startup graph (Figure A-7). Outports have been added for the connection to the memory function and to the load AS0 function. From this point downward, the graphs from the original model are used. Figure A-67 is the subgraph of node **LOADAS0** in Figure A-65. This

graph is cut from the upper portion of the second column of Figure A-7. Graph in and outports have been added for the connections to preceding and following columns on Figure A-7. Again, the hierarchy below the nodes on this graph is that of the basic model.

The remainder of startup would be modeled in the same way.

Figure A-68 is the new subgraph of node **SYSTEMESSAGES** on the top-level graph, (Figure A-63). This graph is divided into four sections, each representing a cluster with its initiation inputs from startup and its termination inputs from failure or shutdown. A cycle between the cluster supervisors of clusters 0, 1, and 2 and the system supervisor in cluster 3 handles the ping and ping acknowledge. One node, **CLUSTER3**, has been expanded in Figure A-69. This graph is basically one half of the system messages graph in the basic model. The other clusters would differ only in having only one inport and one outport to another cluster (**CLUSTER3**) instead of three for the ping and ping acknowledge. The hierarchy below these graphs is the same as for the basic model.

Referring again to Figure A-63, changes similar to these just discussed would be needed for the other portions of the model. An expanded set of LPUs would expand the graphs of nodes **NORMALOPERATIONS**, **SENSORS**, and **PILOTINPUT**. This might well require a grouping of related processes in the first level with the detail pushed down further as we showed for **STARTUP** and **SYSTEMESSAGES**. One might also con-

sider moving the failed and restarted LPUs, or just the restarted ones, into a separate node. If LPUs that access files into system or cluster memory were included in an expanded model, addition of hierarchies to represent the BTBIM memory functions similar to those in startup and reconfigure would need to be incorporated. If more than one failure is desired, one would either expand or duplicate the failure through reconfiguration chain. Shutdown would be expanded in a manner similar to that done for startup and system messages.

Finally, if one determines through a combination of calibration and simulation runs on an expanded model (as is the case using the demonstration 3 scenario) that the PI-bus interface units do not have a significant impact on performance or timing of events the simulation time can be reduced by essentially "shorting out" all of the diamond shape message graphs, (see Figure A-21). This would be accomplished by changing the *node_class* attribute of the parent node (on Figure A-20) to "leaf," ensuring the parent node is mapped onto the proper PI-bus, and setting its firing delay to that used in the subgraph. This can be done throughout the model. The result is that for each of these changes, the simulation software has only one node to track and schedule rather than five. If for some other analysis the user wishes to reincorporate the detailed subgraphs, all he needs to do is change the parent node's *node_class* attribute back to "internal." The subgraph will then be expanded and all other attributes of the parent node ignored.

6. Conclusions

In Paragraph 1.1 it was stated that this modeling effort was the first phase of a two-phase effort. Development of this model was to be followed by validation (or calibration) against actual measurements of performance of the AARTS demonstration 3. Validation was to be followed by expansion of the model to a full PAVE PILLAR mission applications architecture to be used to investigate data transfer functions. The delay of demonstration 3 (and completion of AARTS development) prevents validation of the model against measurements of the actual system. (The system does not yet exist). However, the model has been continuously validated during construction through briefings, technical interchange and demonstrations with the AARTS developer. The principle measure of a model of a system still under design is how well (at any given time) the model represents what the designers visualize as their finished product. Since design decisions continue to be made or revised, this presents a "moving target" that requires frequent, if not continuous, interchange between the modeler and the developer. That interchange was achieved in this effort with the graphical presentation of the ADAS model facilitating communication. The results of the modeling effort have highlighted the value of developing an executable simulation of a system as the design and development progresses. In order to develop a model that can be executed, it is necessary to pursue the implications of design decisions beyond the boundaries of the specific development effort (interface with other systems, hardware etc.). This broader view of the system provides insights into

potential clashes or mismatches across these interfaces that can lead to expensive and time consuming corrective action if not discovered until late in the development cycle. These insights often lead to design changes that "invalidate" the model version that predicted the need for the change. This requires alteration of the model and revalidation in the context of the modified design. This was the course followed in developing the basic model as delivered.

The following are some examples of the benefits obtained from this modeling effort, beyond that of producing a model to serve as a basis for further expansion and analysis of Advanced Systems Architectures. These examples highlight the value of simulation of a component (AARTS) during design of that component in a system (AARTS, LPUs, plus VAMPs) context.

Early in the modeling effort ADAS graphs were developed for the initial loading of modules that were very much like the current model of passive loading of AS0 software. The modeling was based on appendix G to reference 17, which is the description of SUROM Processing developed by Westinghouse Electric Corporation, the developer of the VAMP hardware. Following this document the ADAS graphs assumed that after completion of BIT a module repeatedly places a two-word "ready" message onto the PI-bus at regular intervals. The loader (in this case the BTBIM), when ready to load the module, responds with a four-word message (word count, 1750A Data Destination Address (DDA), 1750A Execution Start Address (ESA), and Expected Checksum). Upon receipt of this message, the SUROM software sets

up the PI-bus interface unit to point to the DDA, receives the bootstrap loader, checksums the load, and transfers control to the ESA. Then the bootstrap loader proceeds to download the AARTS software. When our approach was communicated to the AARTS development team we were informed that the AARTS design would not include a bootstrap loader. The design approach was to load the AS0 software directly following the four-word message. We challenged this approach, pointing out that the reference limits the maximum word count in the four-word message (and presumably the download) to 52K. After also recognizing that the one word (16-bit) word count would not accommodate the AARTS file size, we were informed during "final" demonstration of the startup model that the bootstrap loader was, in fact, necessary. The ADAS graphs were redone. This example highlights the fact that the modeling effort itself benefits and supports the design process. Modeling with an architecture tool such as ADAS can identify problems at the interface of the system being designed early in the design phase (had this modeling effort started earlier, this problem would have been discovered earlier) rather than, as is too often the case, not until integration testing.

During the effort to calculate firing delays for the model it was found that the PI-bus data transfer rate we were instructed to use was five times faster than the assumed memory access rate. We could find no indication of a buffer in the PIBIU and this was confirmed. The model, as it now stands, includes a small buffer in the PIBIU, a faster memory access rate, a lower PI-bus transfer rate, and double-word access (32

bit) between PIBIU and memory. This is a problem, we are informed, that is to be corrected in this manner by WEC, the hardware developer.

Both of the preceding examples can be summed up in a single observation: Development of an executable ADAS model of an ongoing design brings to the designers an increased awareness of constraints imposed by hardware or interfacing software systems.

Finally, some insights into the AARTS design have been derived from the modeling effort and the results of simulation runs. Two have already been mentioned in Chapter 4. These are the dominance of checksum times in software loading and of "detection of failure" in reconfiguration. Both of these may be the result of high estimates of the time consumed by processes that can be speeded up. The checksum time is the result of a simple estimate, 10 instruction cycles per word, and can be refined. In the case of the failure detection, it may be no more than a bad choice of a single parameter, Pulse_Timeout. Table 6.1 shows the value cited for this parameter in five different parts of Reference 11. Three of these are found in a subparagraph titled "Limitations" and two in the discussion of unit processing. They vary from three pulse periods (.375 seconds) to ten pulse periods (1.125 seconds). The latter is used in the model. This can be changed by simply changing the value of the attribute *firing_delay* on node DETECTMISNGPLSE in the subgraph of node DETECT in the AARTS graph. Finally, some concerns have been raised about the behavior of the AARTS itself.

Table 6.1. Values for Pulse Timeout

(Source: Reference 11)

Software Unit	Processing or Limitation	Page Number	Value
SE.Kernal	limitations	24	5 pulses
SE.Cluster Management	text	32	5 pulses
SE.Cluster Management	limitations	37	10 pulses
SE.System Management	limitations	48	10 pulses
SE.System DB Management	text	51	3 pulses

The first concern is with startup sequencing. In the model, as delivered, sequencing has been forced in the hierarchy below the node that represents the SMM functions in the passive loading of the AS0 software. In order to accelerate startup, particularly in a larger system than the one modeled here, one may want to allow loading of multiple modules in parallel. In fact, the sequenced loading has never been confirmed to us as a feature of the AARTS software. It was a "ground rule" we were given to work with. In such a case, one must be sure that the MABIU in a cluster is loaded before any CPU module commences hot backup arbitration. Otherwise one could end up with more than one system supervisor.

A similar concern has to do with when the system supervisor commences configuration (loading LPUs etc.). The model, as delivered, will not commence configuration until all models have loaded and started AS0. If a module fails on startup in the model as configured, configuration will not take place. Since we were instructed to consider no BIT failures, this is not a problem with this scenario. However, some means should

be provided for the system supervisor to determine when to start configuration that allows for failure of one or more modules in the startup process. The algorithm must be such that it does not cause premature loading into a reduced hardware configuration when the full configuration actually does become available. This is not a problem in the demonstration 3 scenario since all of the LPUs could be run in a single processor. The expanded ADAS model of phase II could produce performance assessment of alternative approaches to this feature.

References

[1] *Architecture Specification For PAVE PILLAR Avionics,* Final Technical Report for period Sept. 1985 - Oct. 1986, AFWAL-TR-87-1114, Wright-Patterson Air Force Base, Ohio, January 1987.

[2] E. Schelling, L. McFawn, and D. Williams, *Critical Item Development Specification for the Avionics Bus Interface Module for the VAMP,* SSTA10301, WEC, Baltimore, MD, May 1990.

[3] Society of Automotive Engineers, *Linear Token Passing Multiplex Data Bus User's Handbook,* AIR 4288, Draft Issue 2, Phoenix, AZ meeting of ASD, April 1990.

[4] *Architecture Design and Assessment System (ADAS),* USER Manual, Version 2.5, Research Triangle Institute, Research Triangle Park, NC, 1988.

[5] *Listings of Demo-3 LPU Packages as of that Date,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, July 1989.

[6] *Software TLDD For AARTS ATSD Subcontract (Boeing),* Boeing Advanced Systems, Seattle, WA, December 1988.

[7] *Software Detailed Design Document For AARTS ATSD Subcontract (Boeing),* Boeing Advanced Systems, Seattle, WA, December 1988.

[8] *Interface Requirements Specification for the AARTS,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, May 1988.

[9] *Interface Requirements Specification for the AARTS,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, January 1990.

[10] *Software Detailed Design Document for the AARTS (Without Section 3),* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, September 1989.

[11] *Software Detailed Design Document for the AARTS (Section 3),* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, January 1991.

[12] *Software Requirements Specification for the AARTS,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, January 1989.

[13] *Software Requirements Specification for the AARTS,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, January 1990.

[14] *System Segment Specification for the AARTS,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, September 1987.

[15] *System Segment Specification for the AARTS,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, January 1990.

[16] *System Segment Specification for the (PCS) for the VAMP,* Westinghouse Electric Corporation, Baltimore, MD, October 1989.

[17] *Functional/Interface Specification for the 1750A CPU,* Westinghouse Electric Corporation, Baltimore, MD, September 1987.

[18] *Functional Interface Specification for 1553B,* Westinghouse Electric Corporation, Baltimore, MD, September 1989.

[19] *Slides from the AARTS CDR,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, May 1988.

[20] *Slides from the Second Quarterly Review,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, August 1988.

[21] *Slides from the Third Quarterly Review,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, January 1989.

[22] *Selected Slides from the Fifth Quarterly Review,* TRW Avionics & Surveillance Group, Dayton Engineering Laboratory, Beavercreek, OH, November 1990.

[23] T.R. Allen. TRW Interoffice Memorandum, Subject: SMM/AARTS Interface, Dayton, OH, April 1990.

[24] J.W. Stautberg. FAX Subject: Summary of LPUs, January 1991.

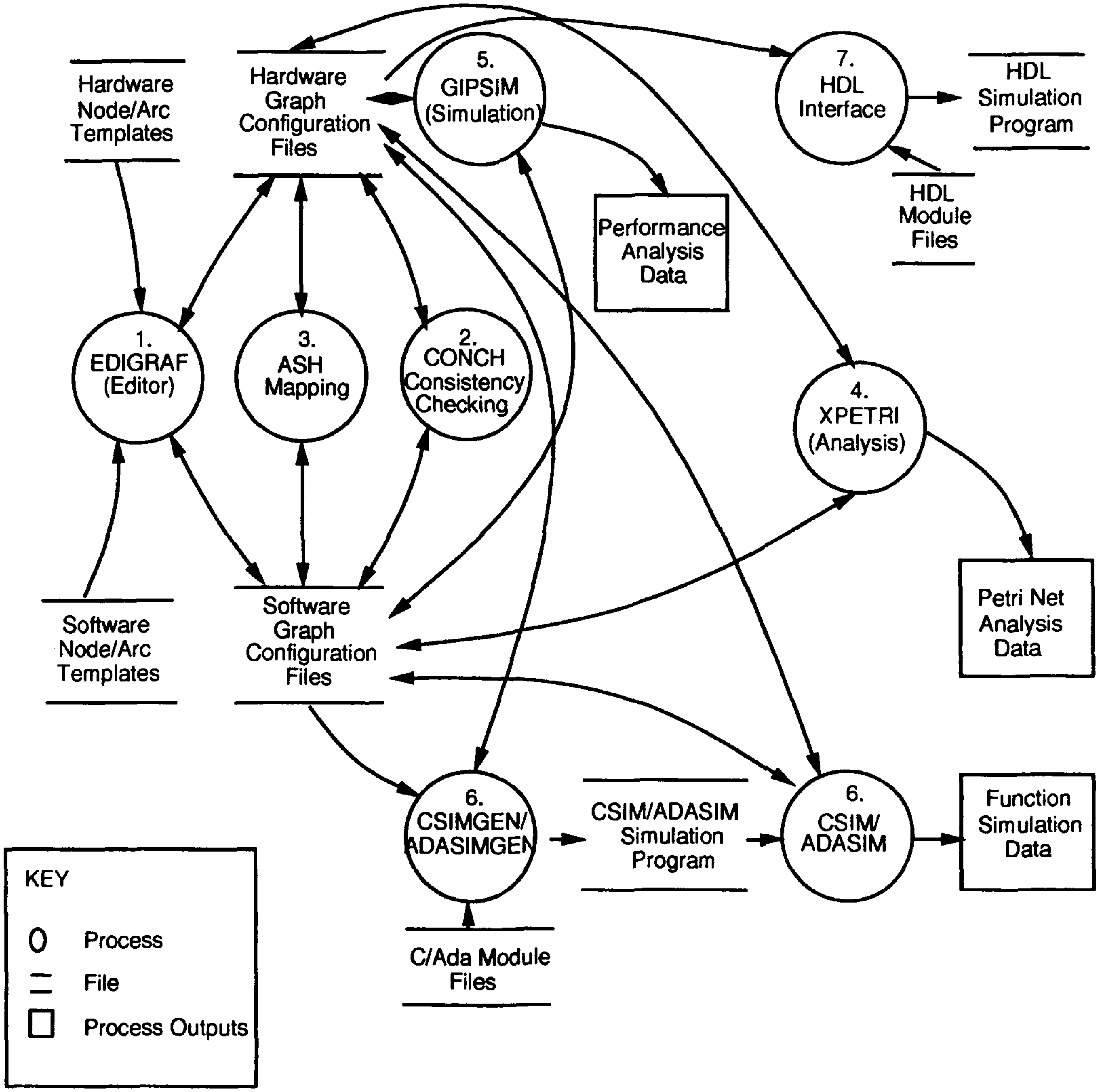

Figure A-1. The ADAS System Configuration

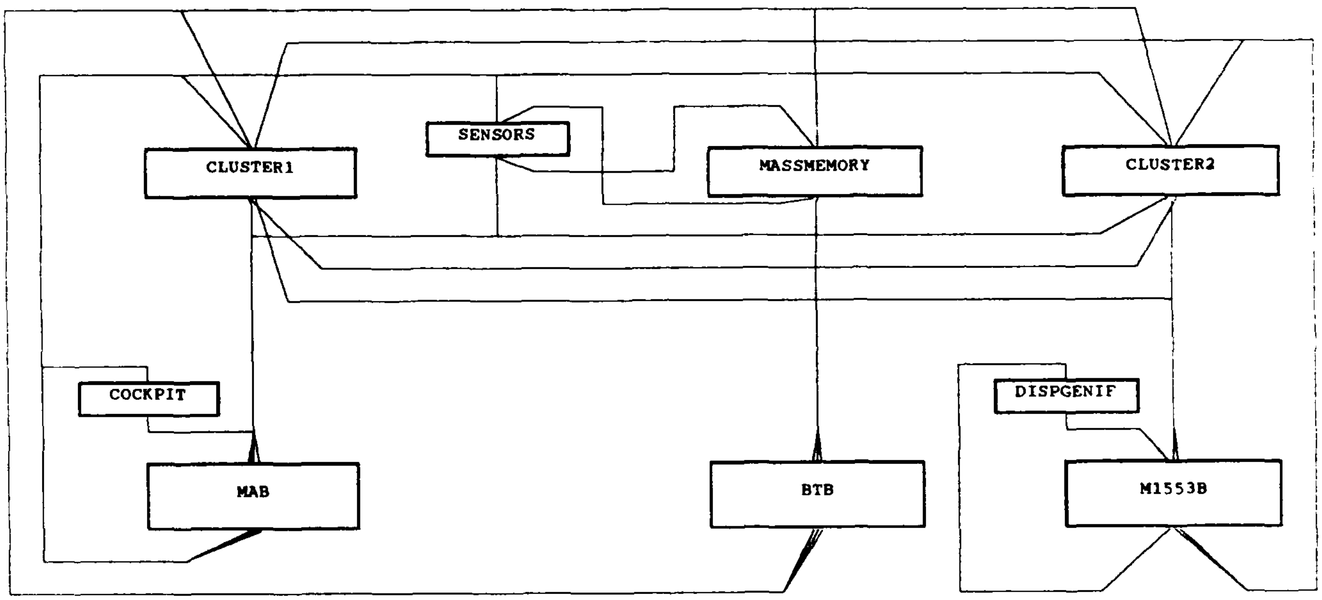

Figure A-2. Top-Level ADAS Hardware Graph

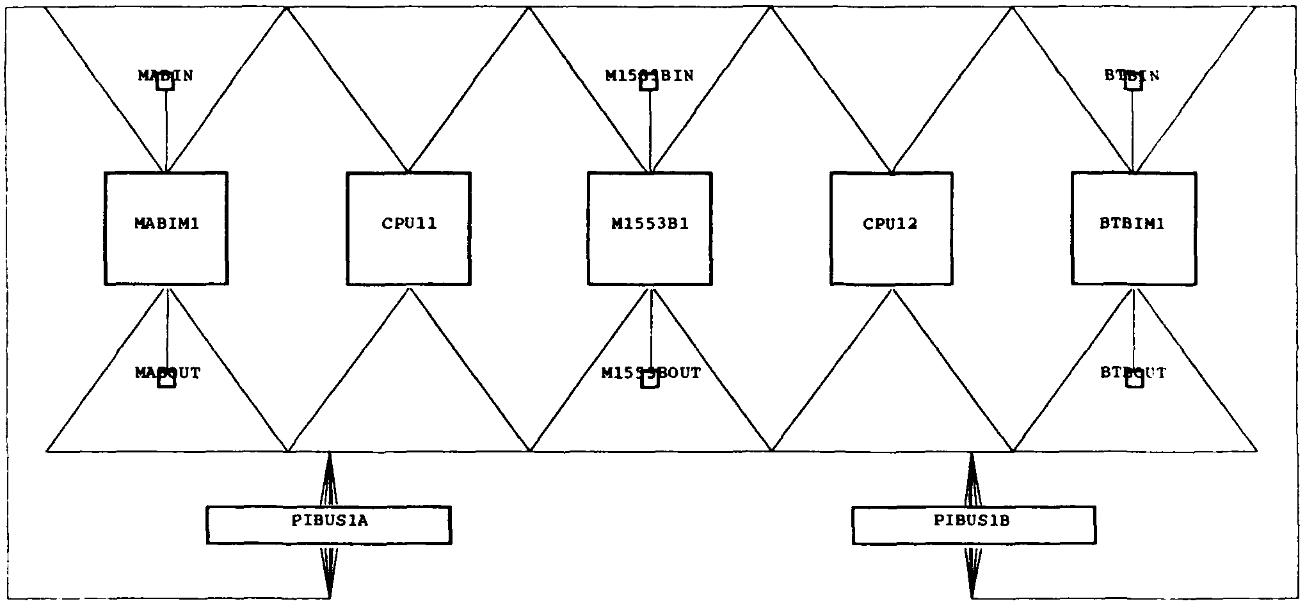

Figure A-3. ADAS Hardware Graph of Cluster 1

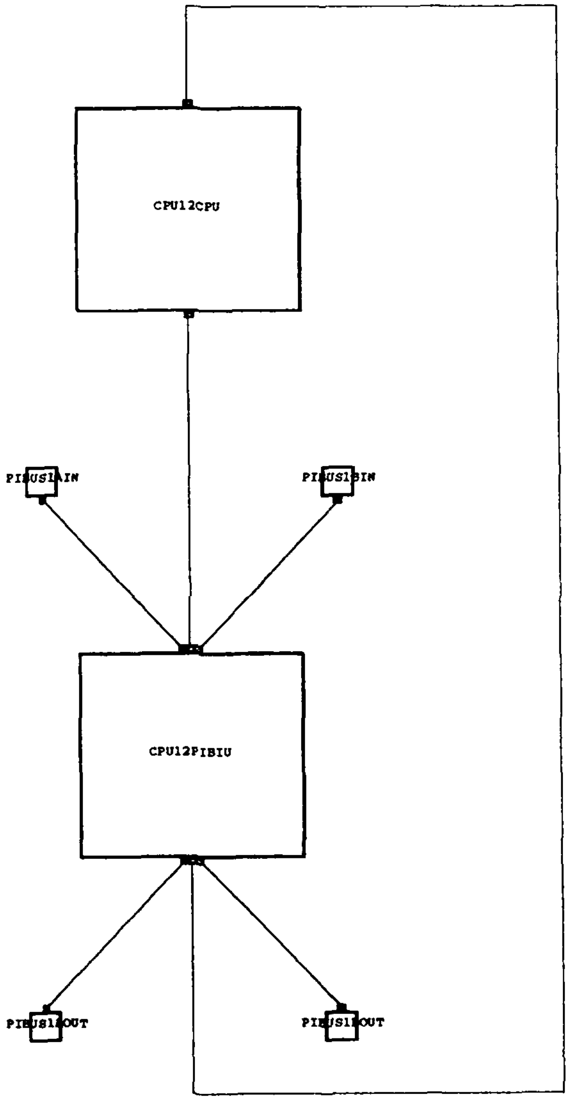

Figure A-4. ADAS Hardware Graph of a CPU Module

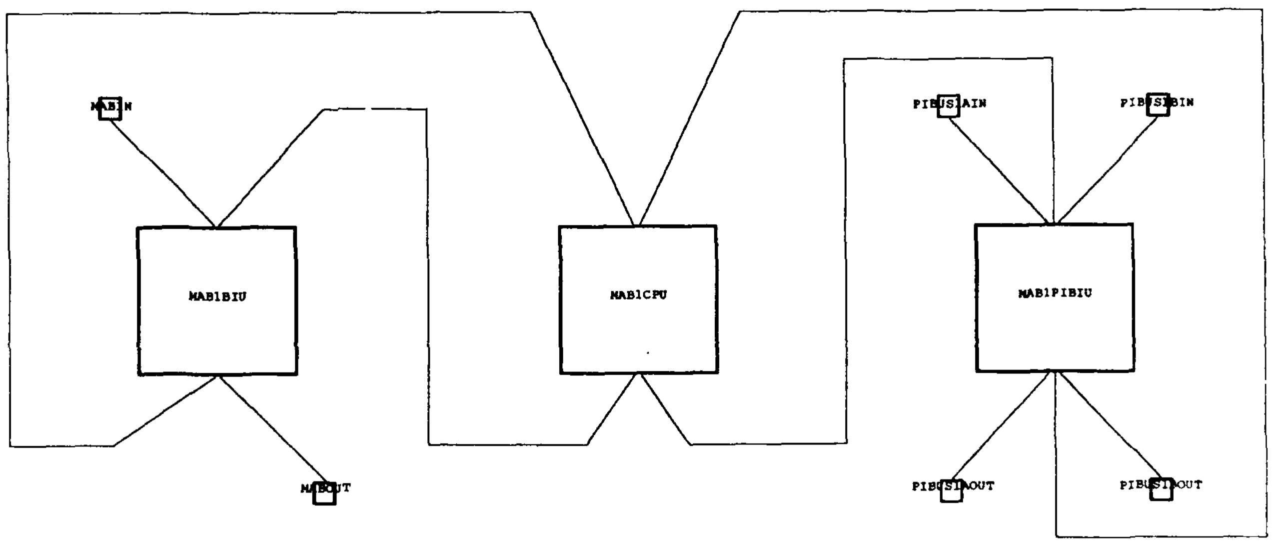

Figure A-5. ADAS Hardware Graph of a Bus Interface Module

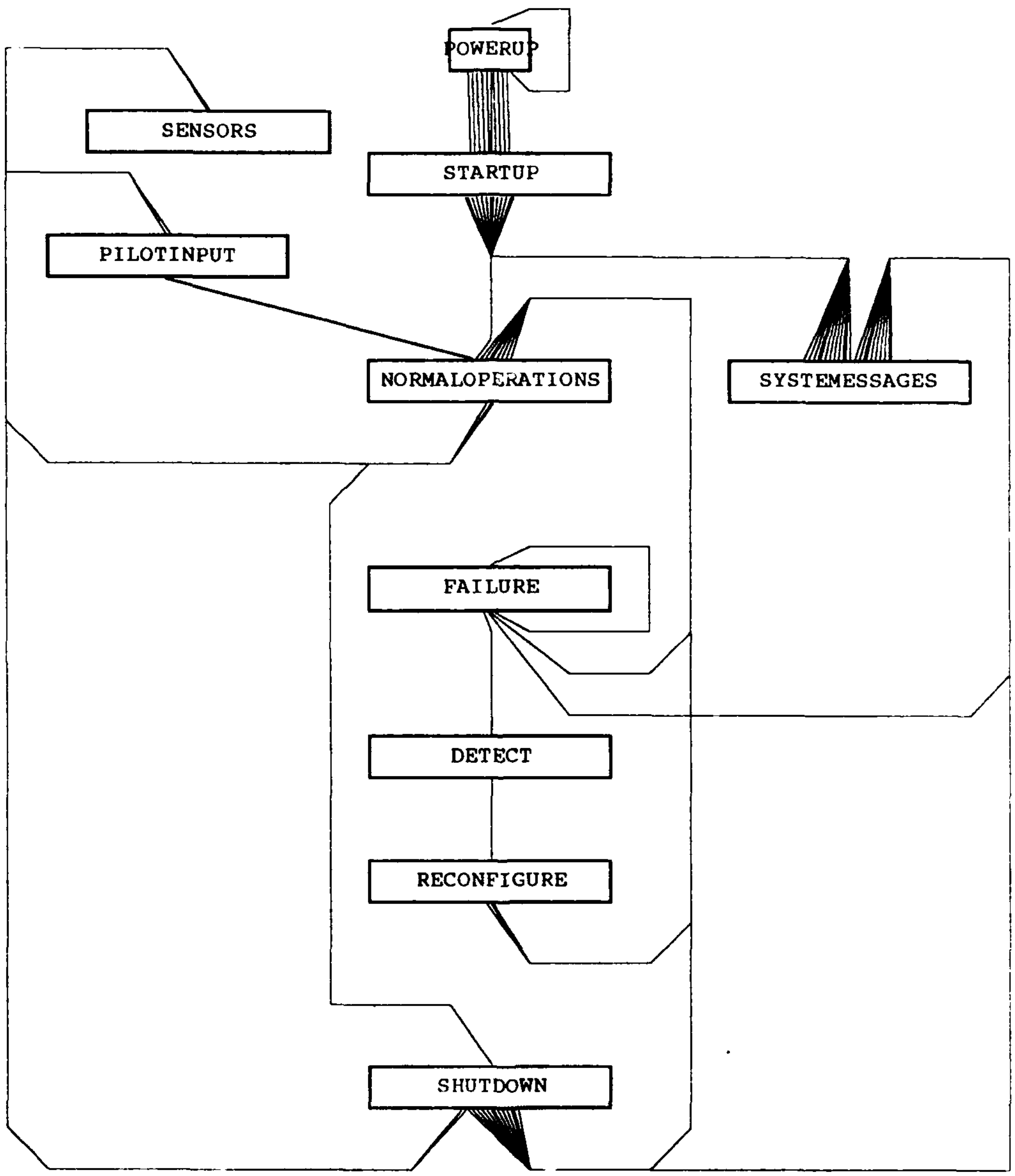

Figure A-6. Top Level ADAS Software Graph

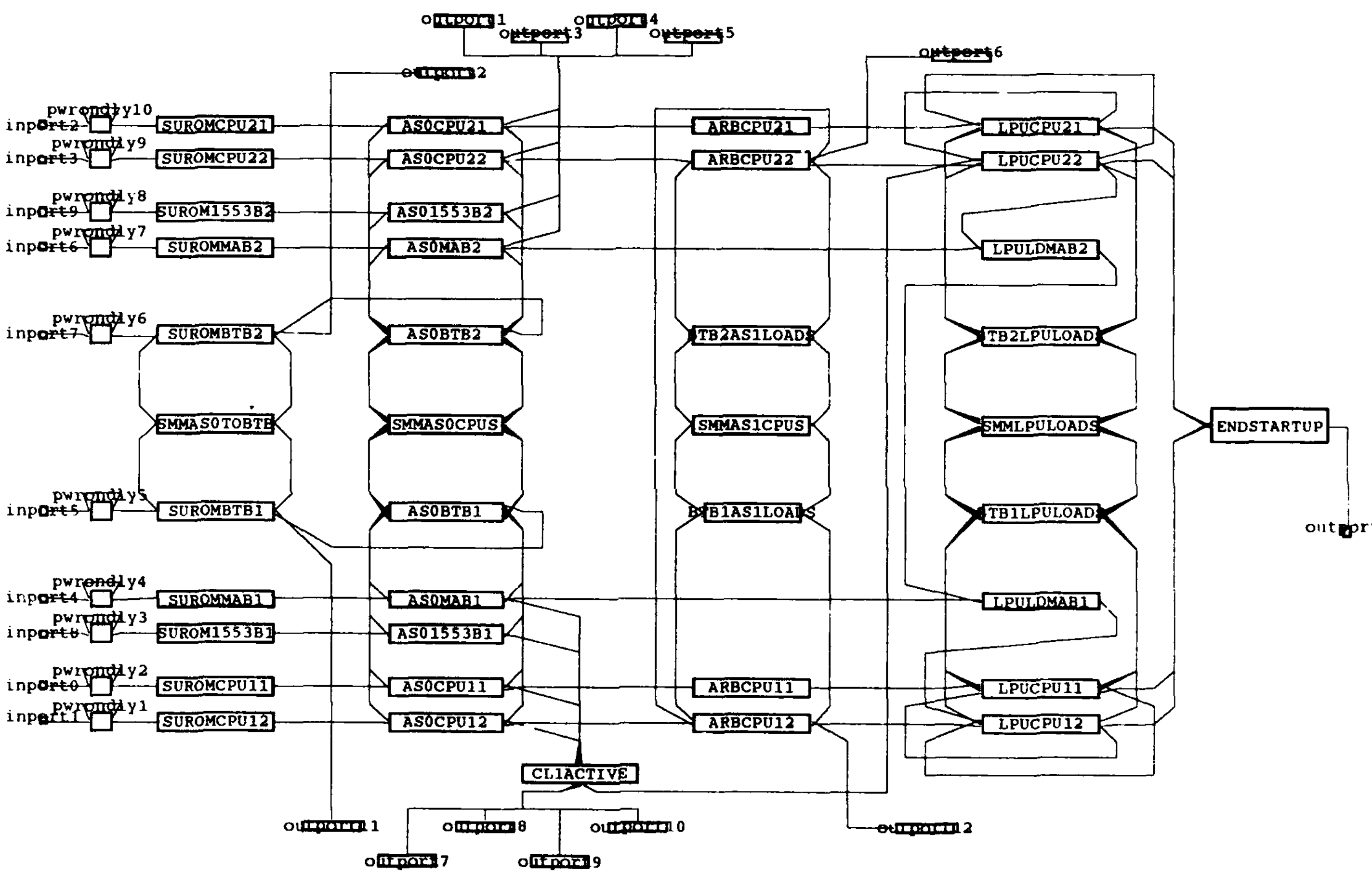

Figure A-7. Startup Graph

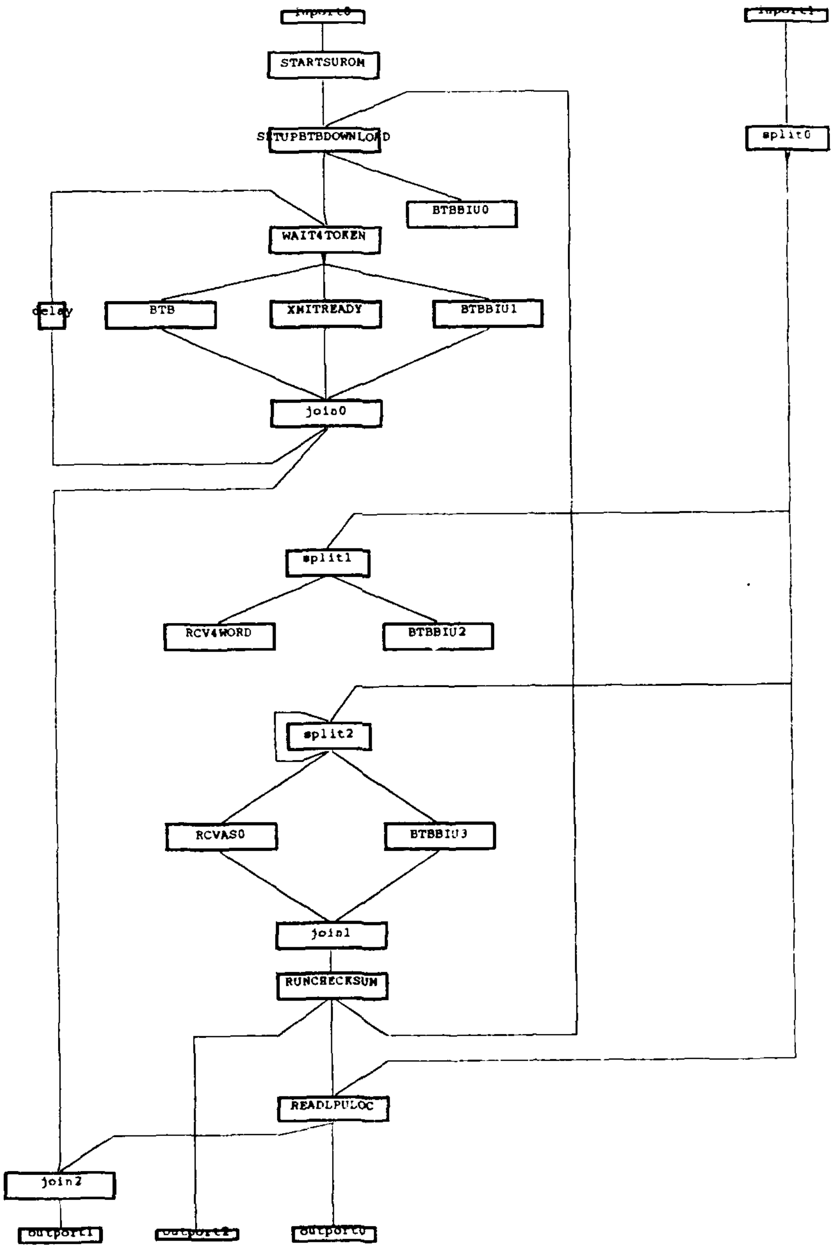

Figure A-8. BTBIM Active Load

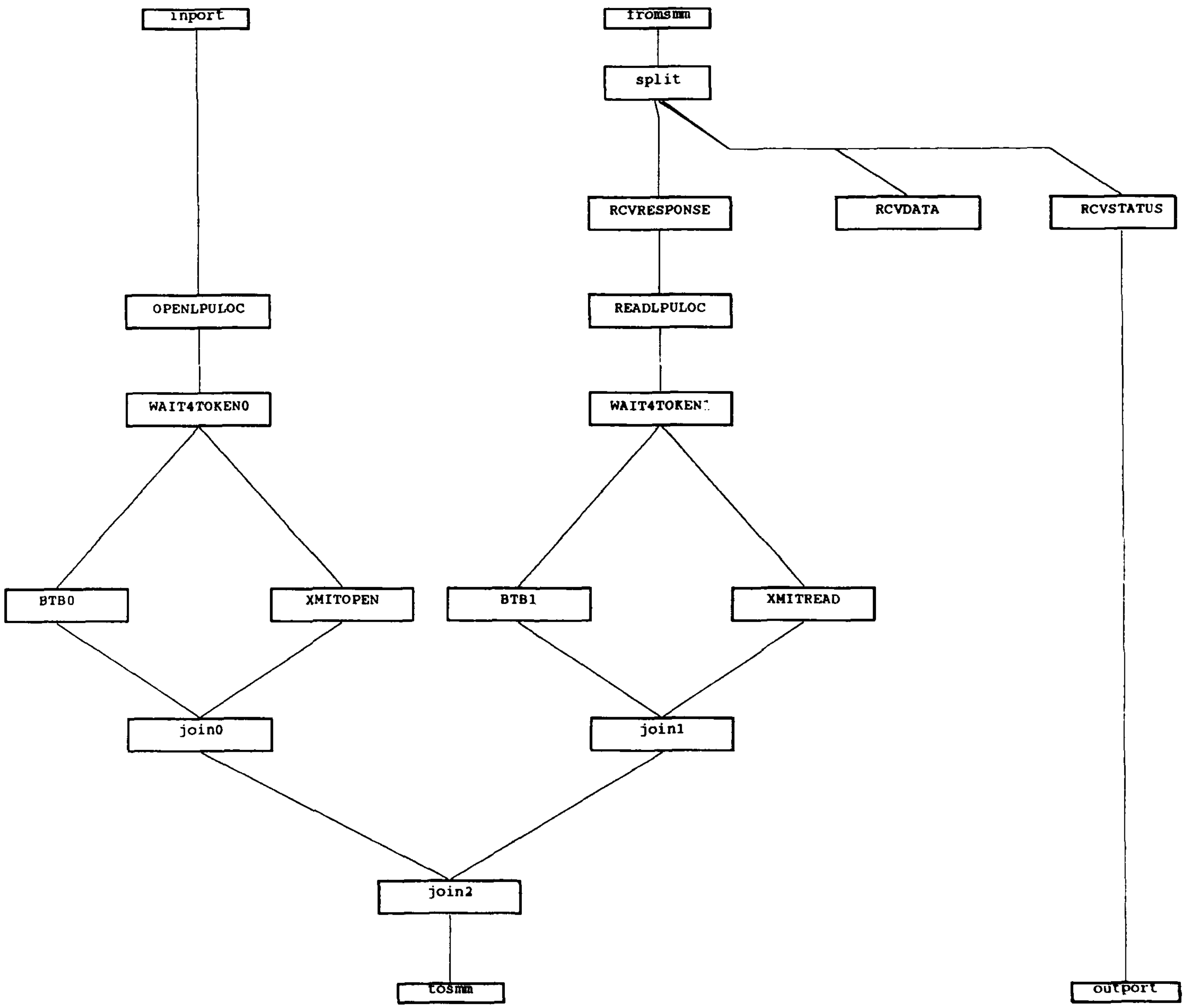

Figure A-9. BTBIM Read LPU_ATTRIBUTES File

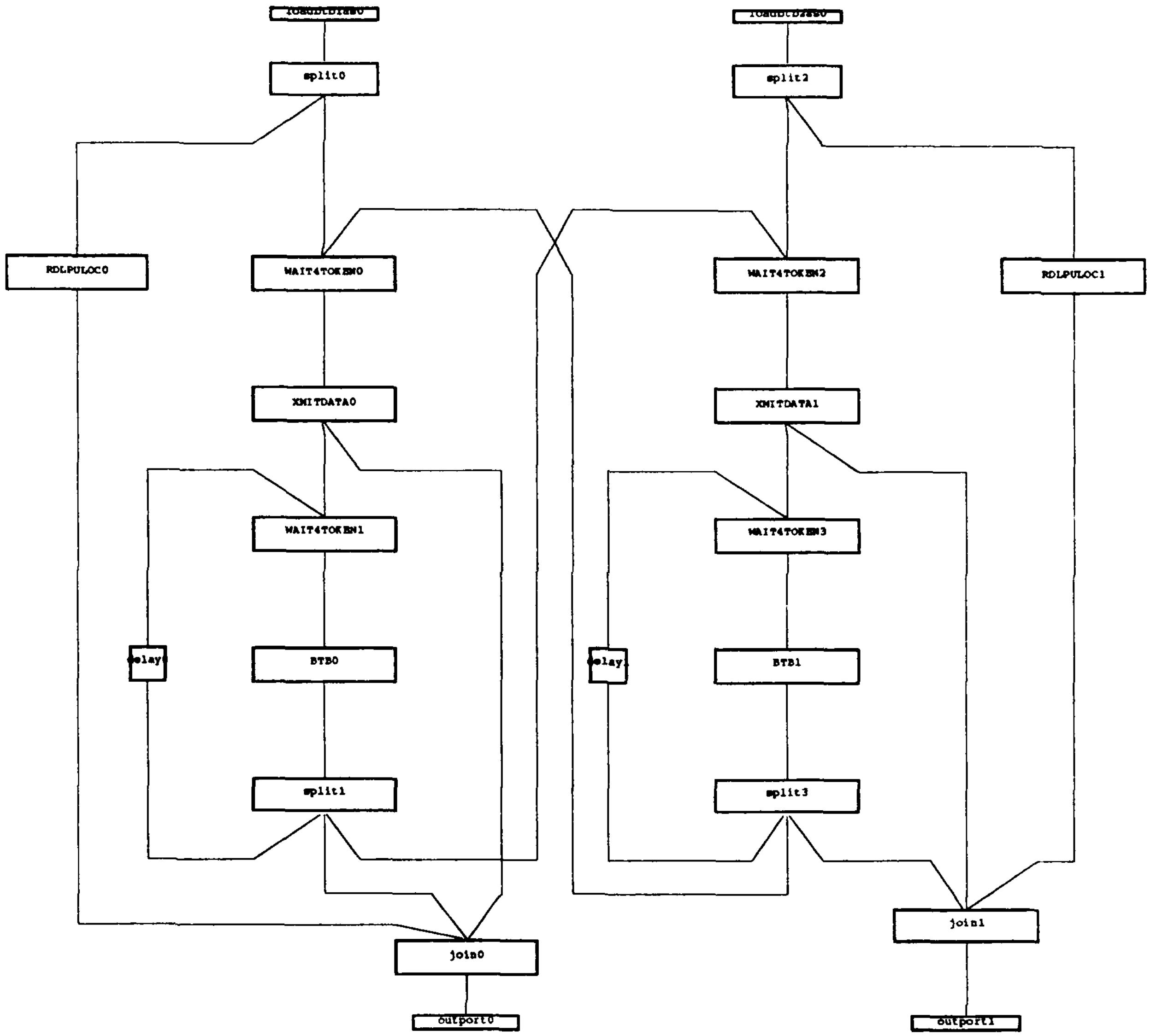

Figure A-10. SMM Load AS0 into BTBIMs

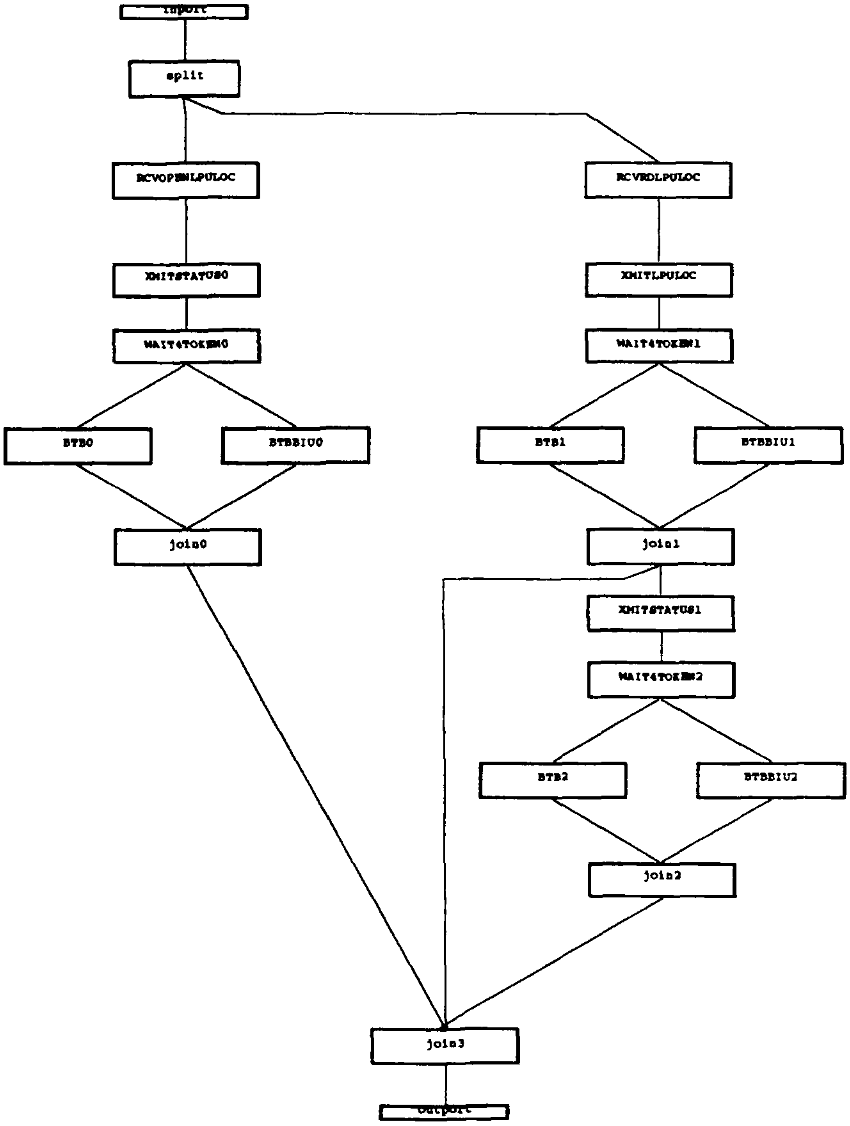

Figure A-11. SMM Load LPU_ATTRIBUTES File into BTBIM

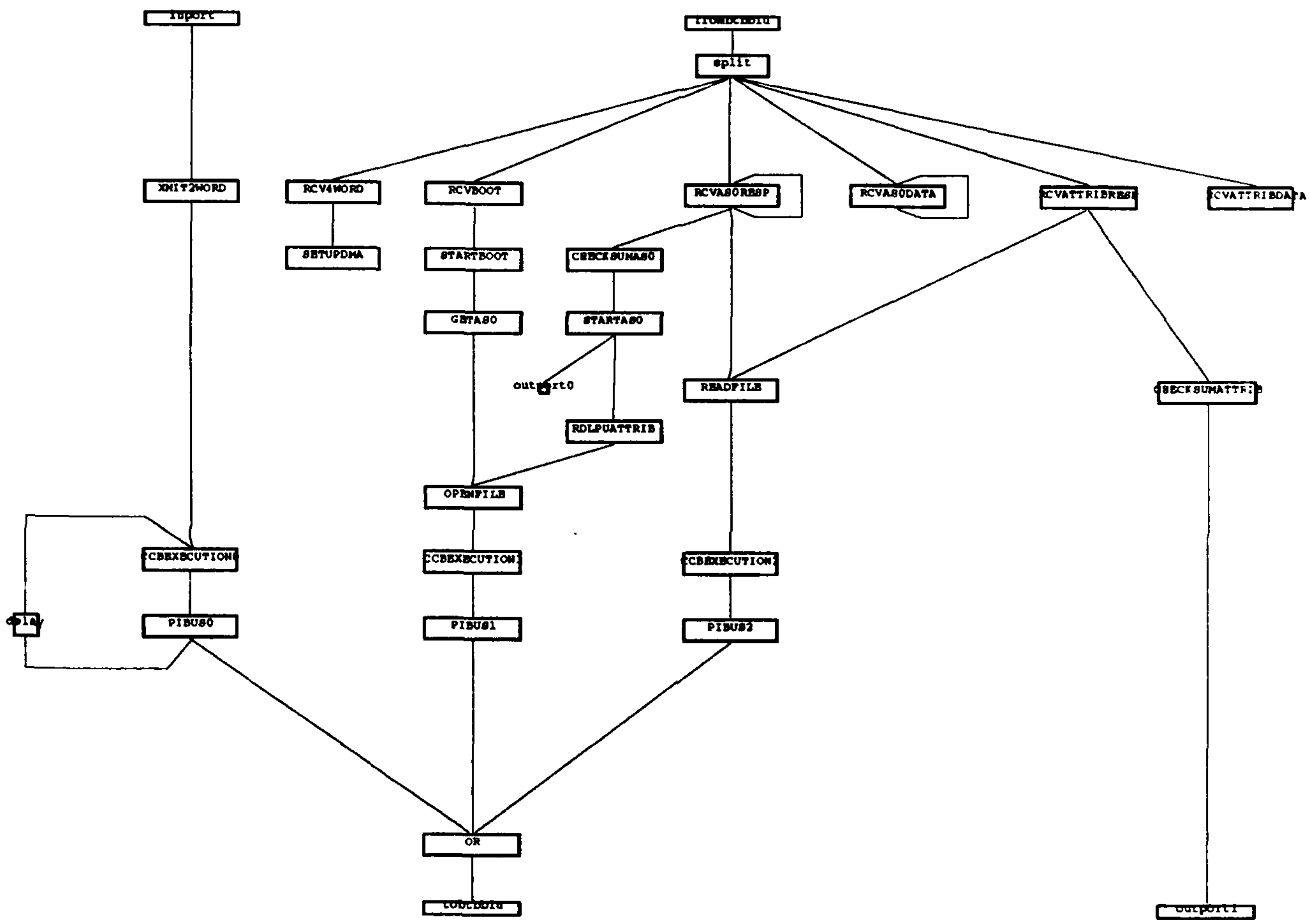

Figure A-12. CPU Receive BOOT and AS0 Load

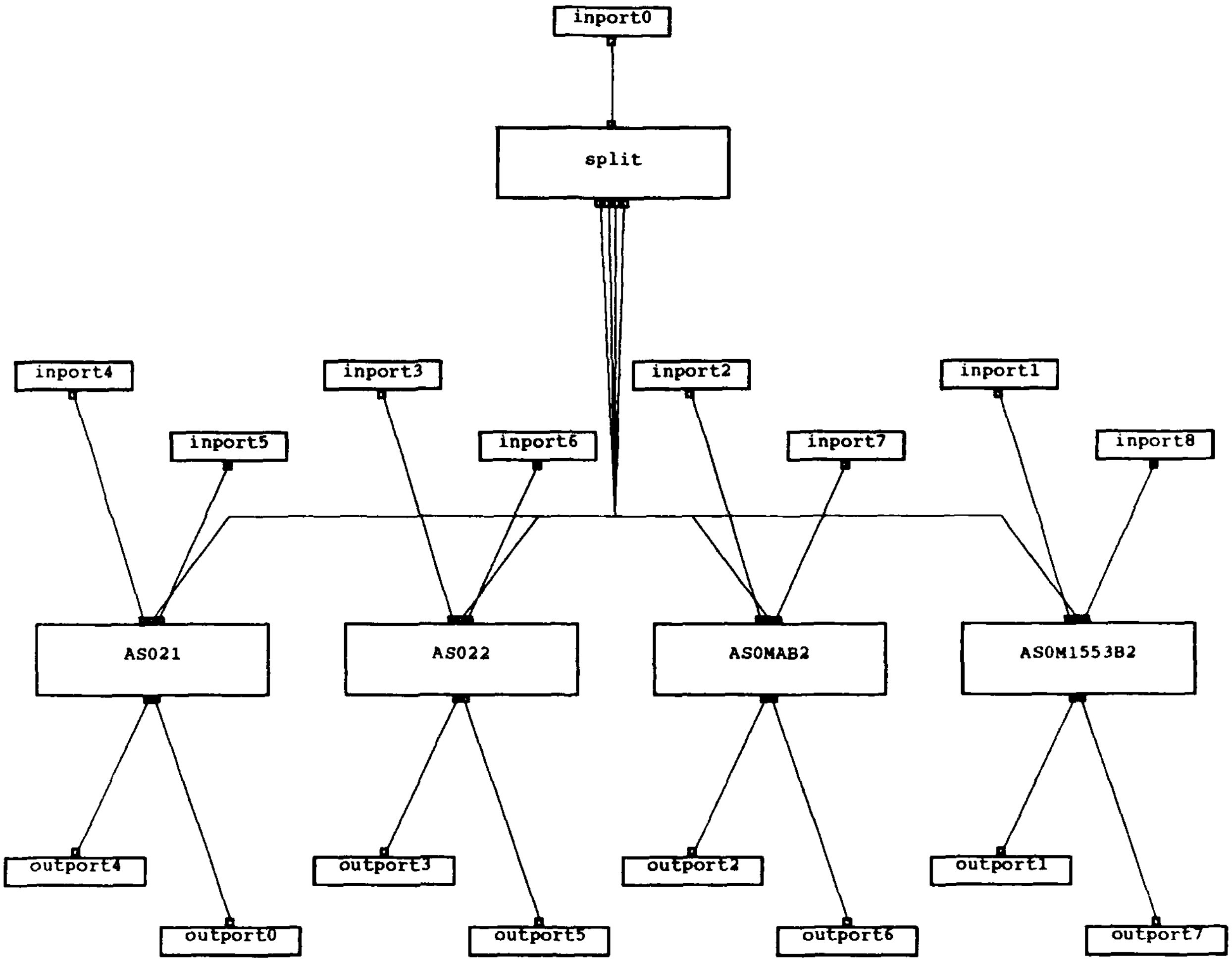

Figure A-13. BTBIM Conduct Passive Load of Clients

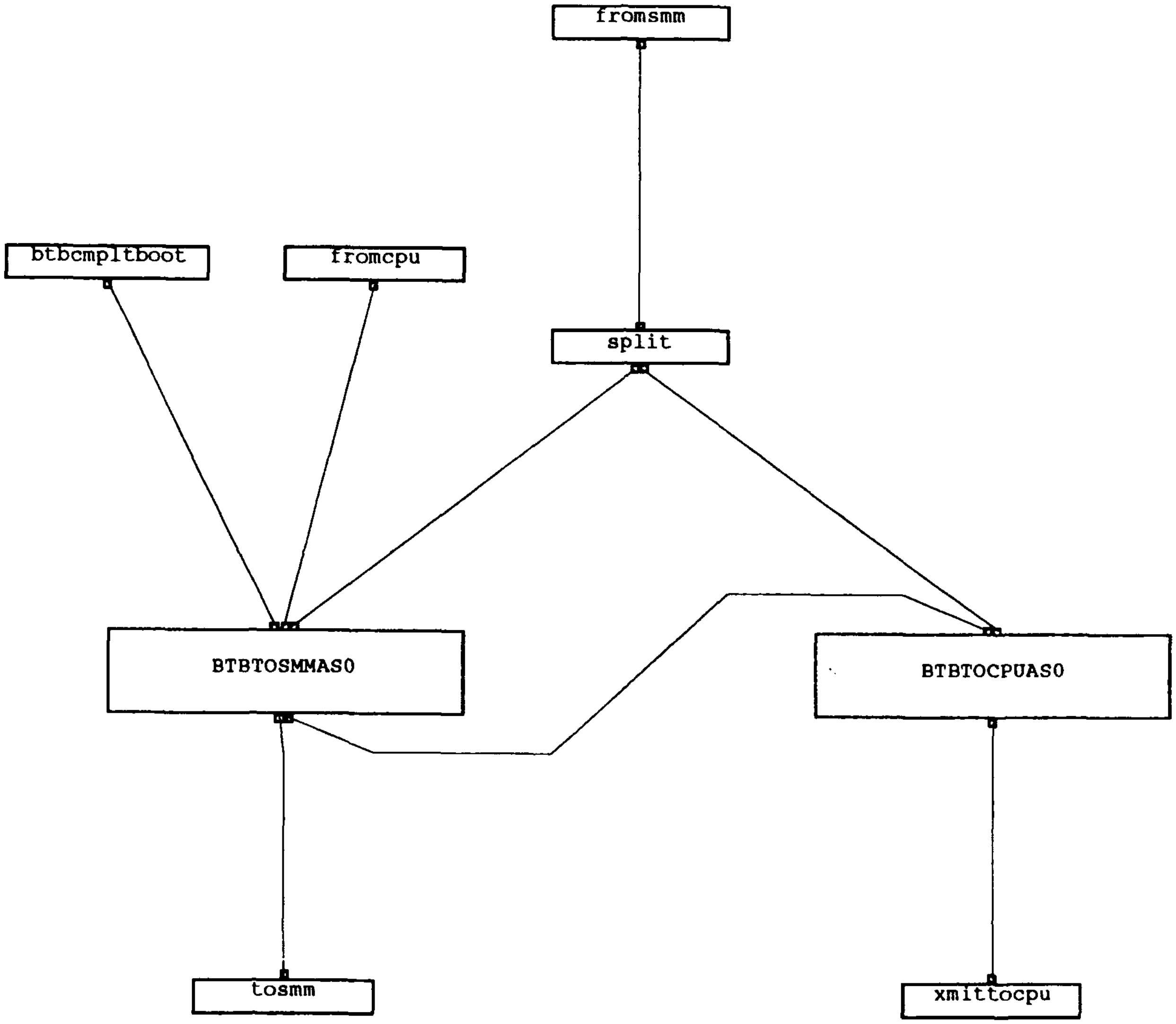

Figure A-14. BTBIM - Client Level

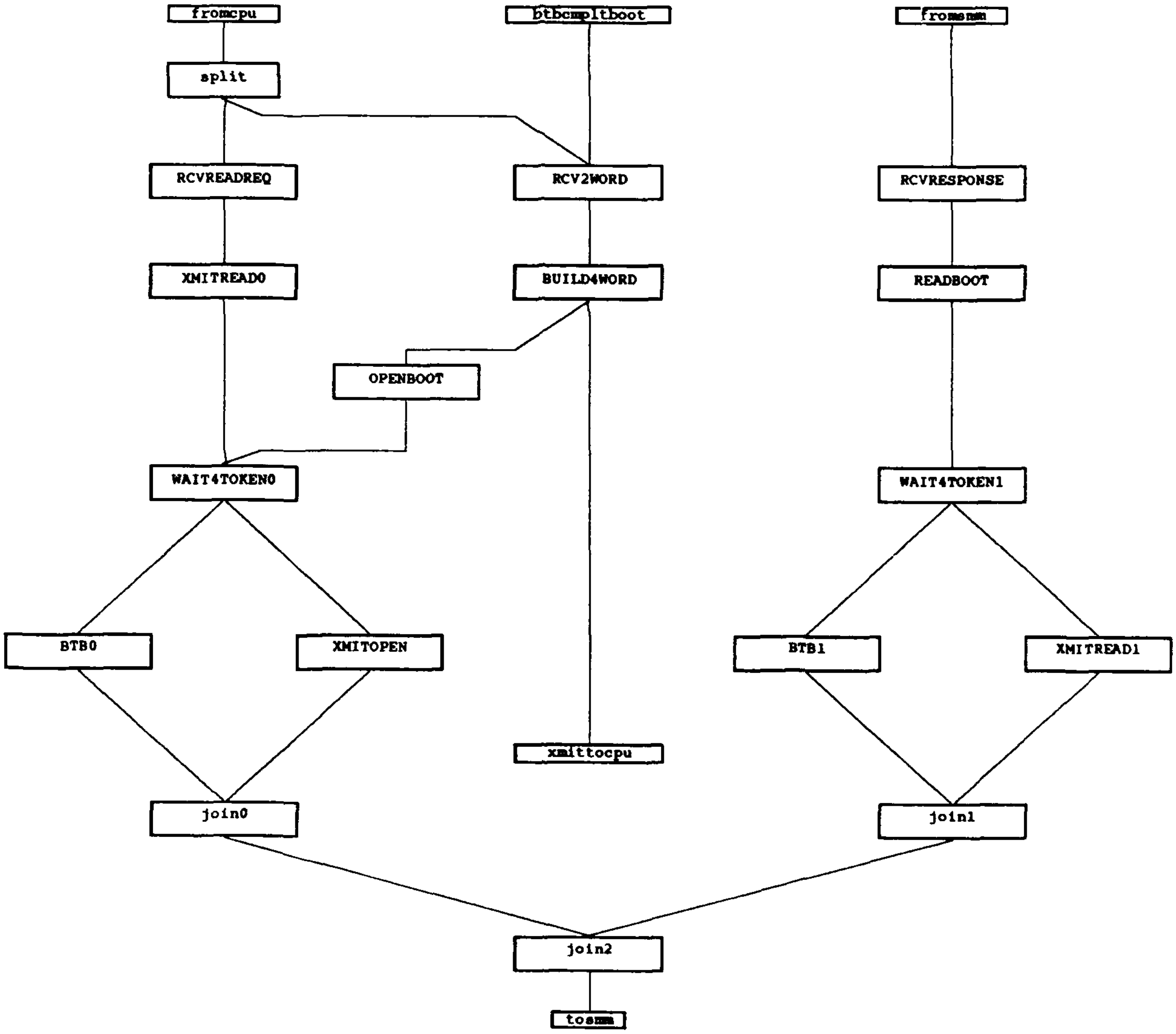

Figure A-15. BTBIM to SMM

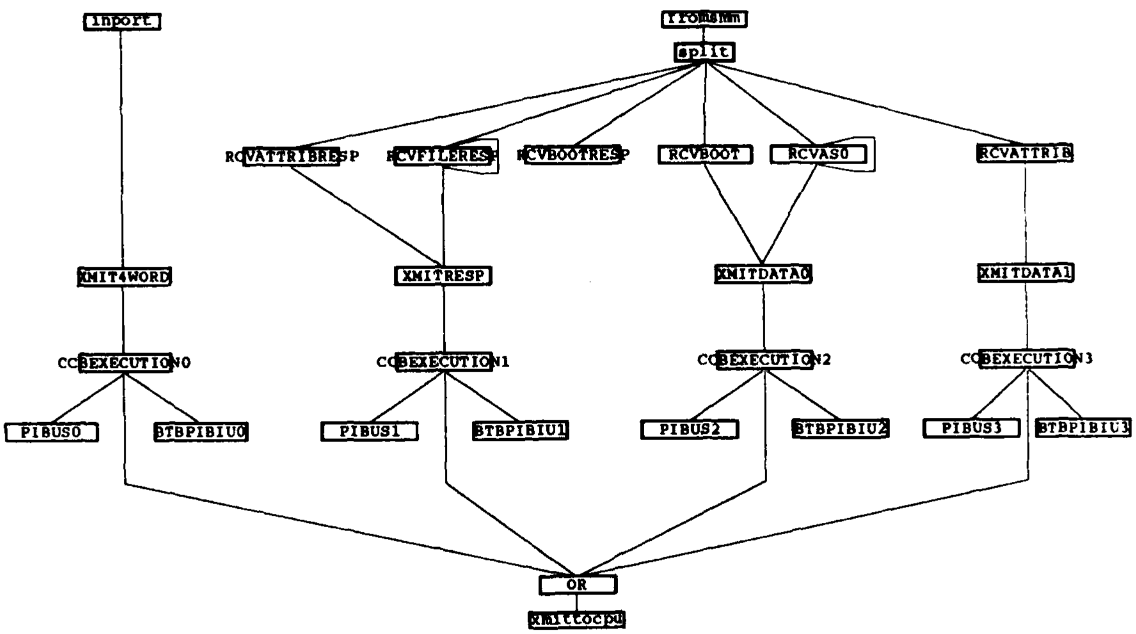

Figure A-16. BTBIM to Client

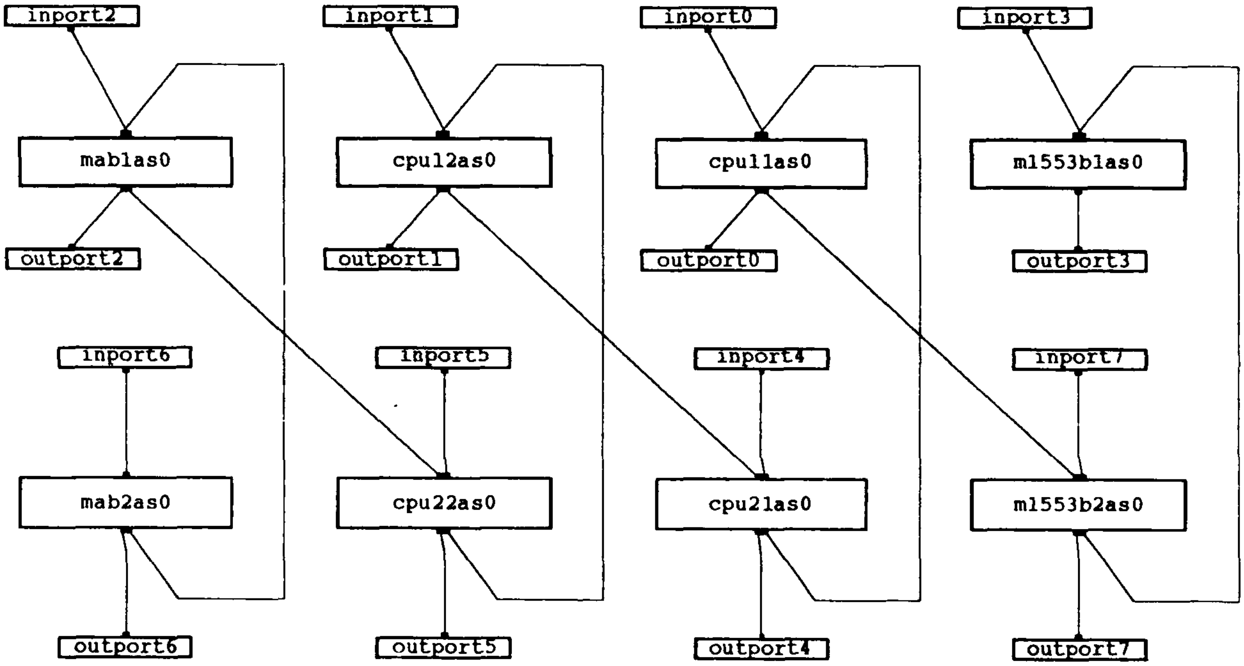

Figure A-17. SMM Passive Loading of 8 Modules

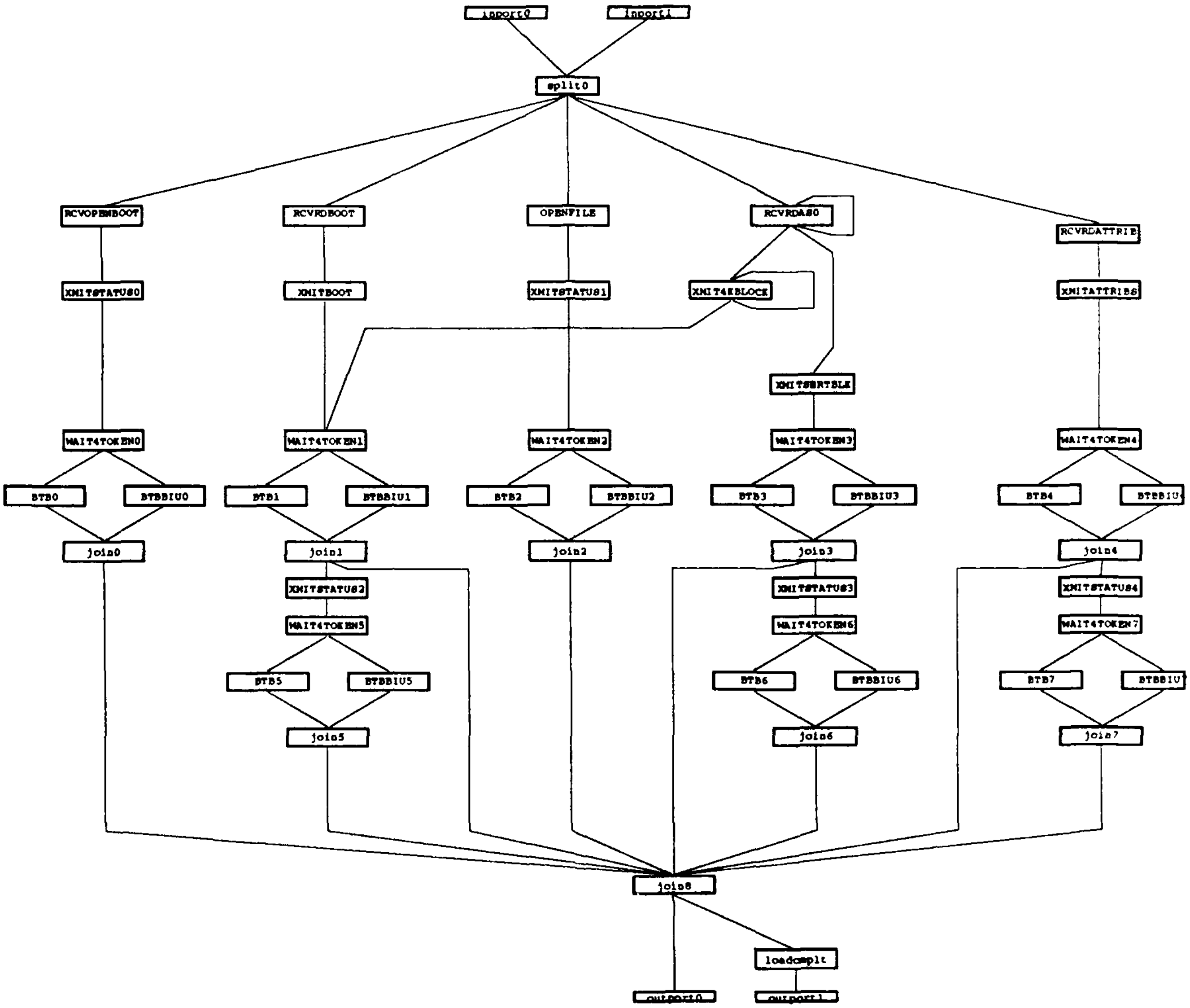

Figure A-18. SMM Passive BOOT and AS0 Load to a CPU

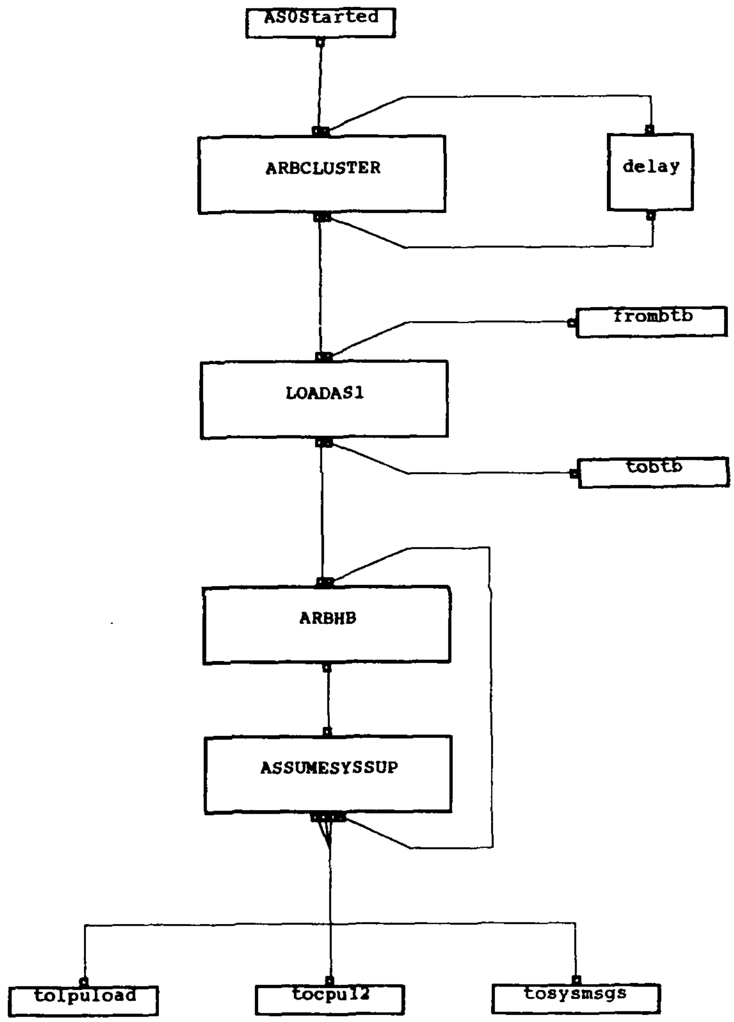

Figure A-19. Arbitration by the Winner of the System Supervisor Role

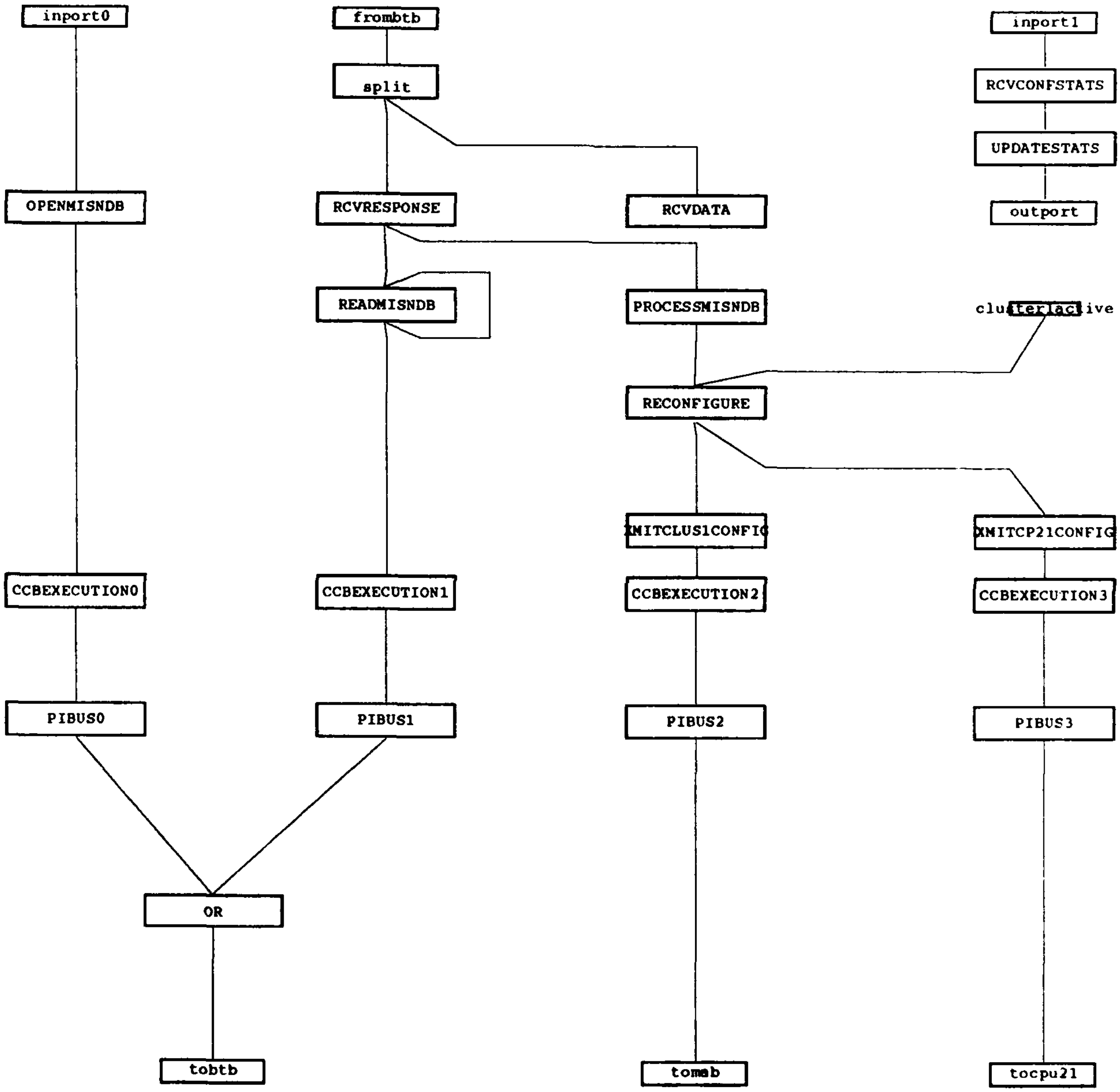

Figure A-20. System Supervisor Load LPUs

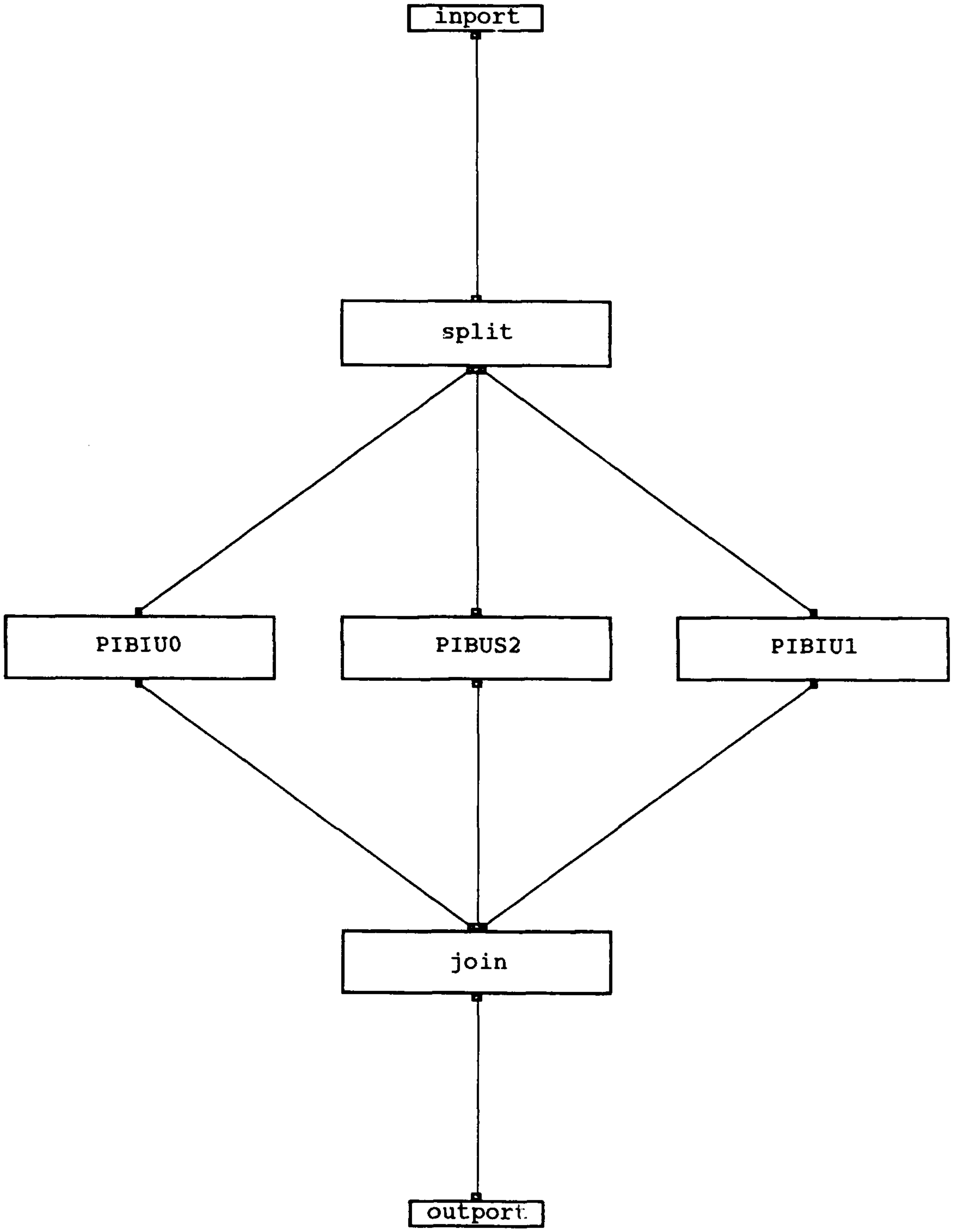

Figure A-21. PI-bus Transmission

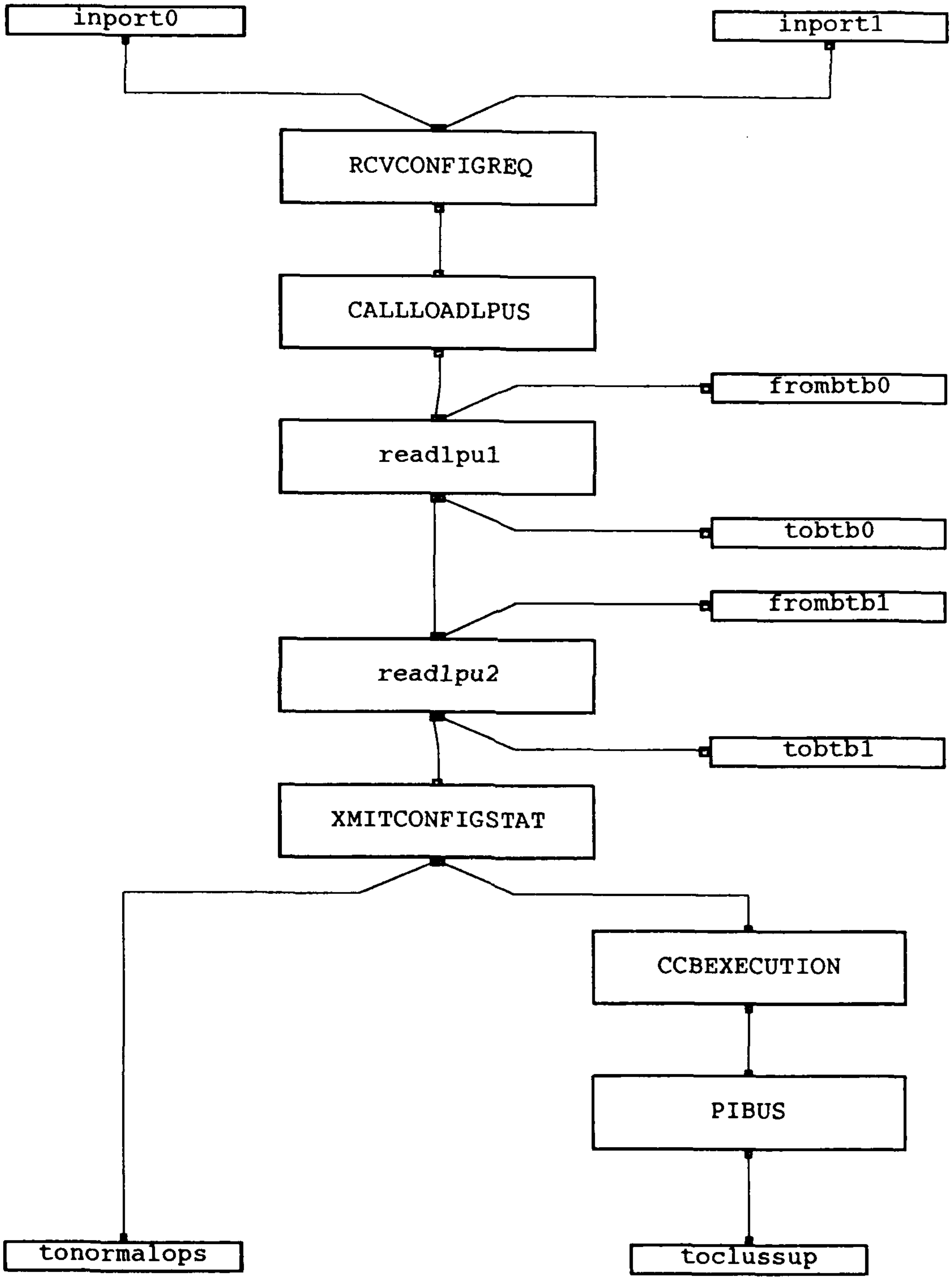

Figure A-22. CPU Load Two LPUs

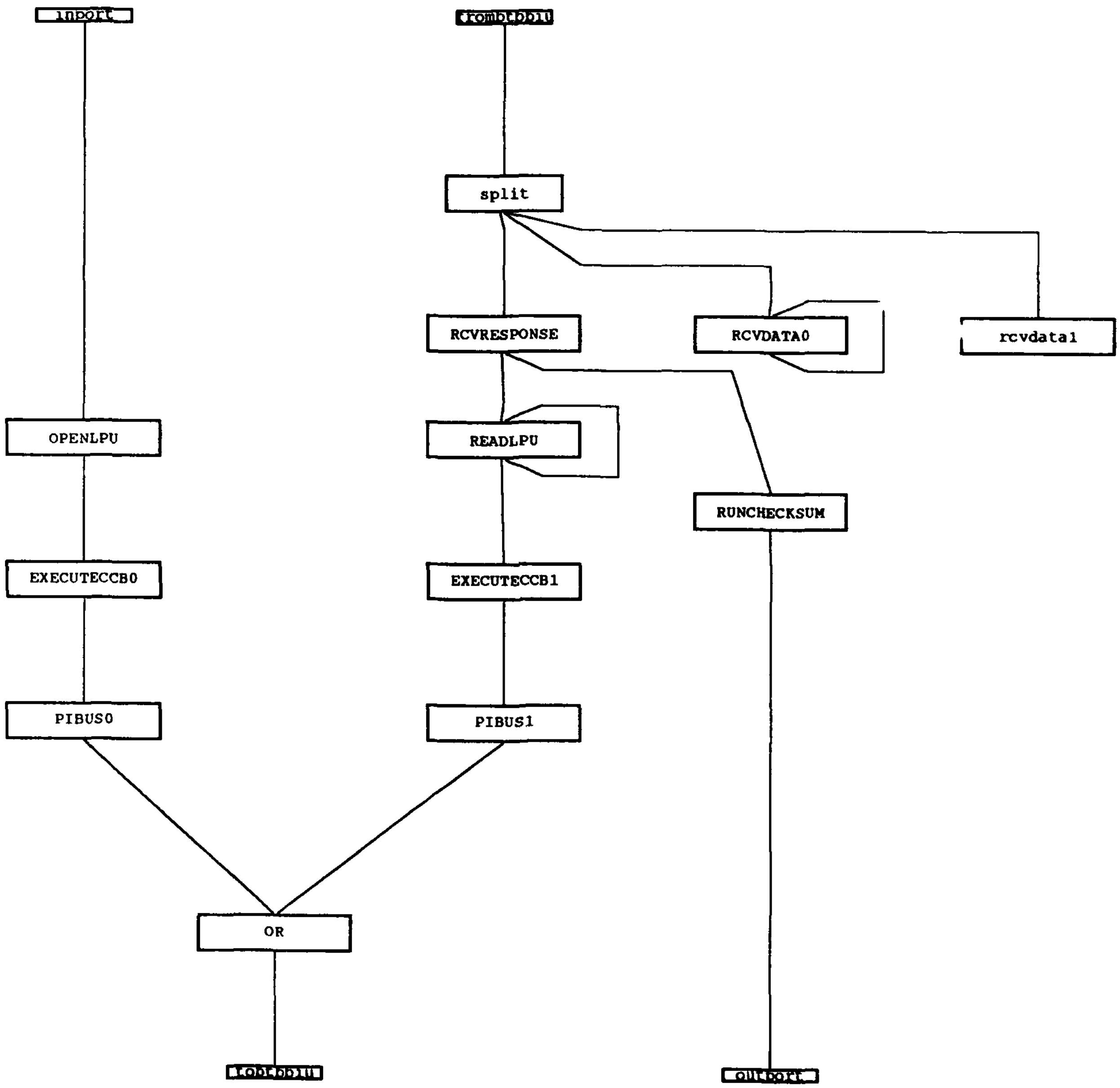

Figure A-23. CPU Load on LPU

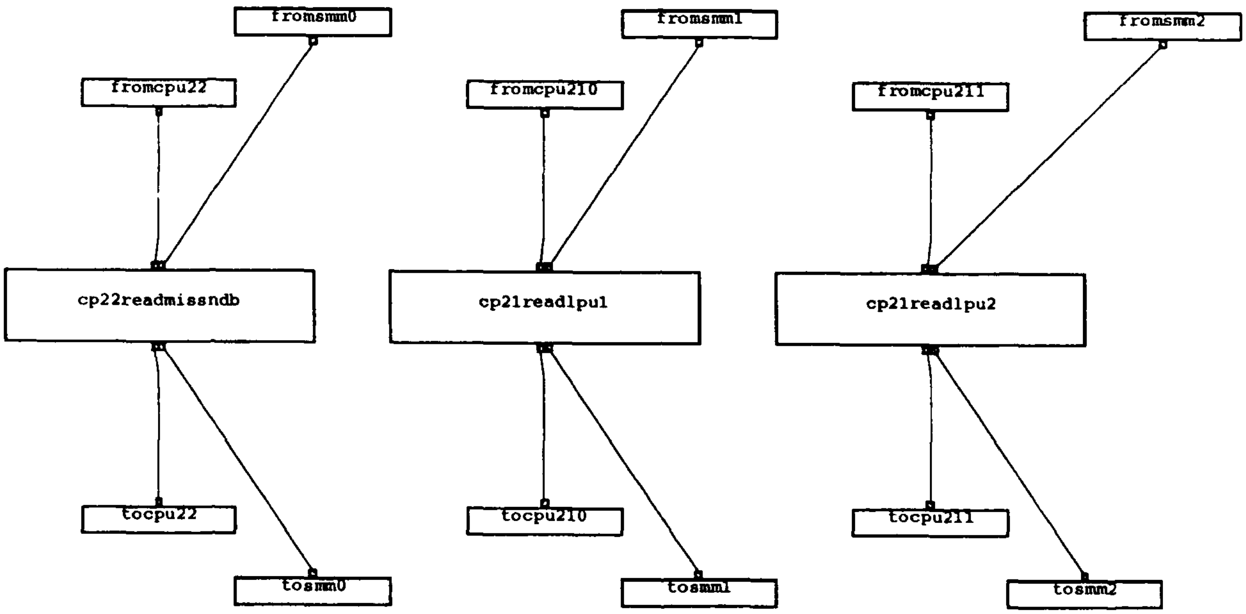

Figure A-24. BTB PASS Tree LPUs

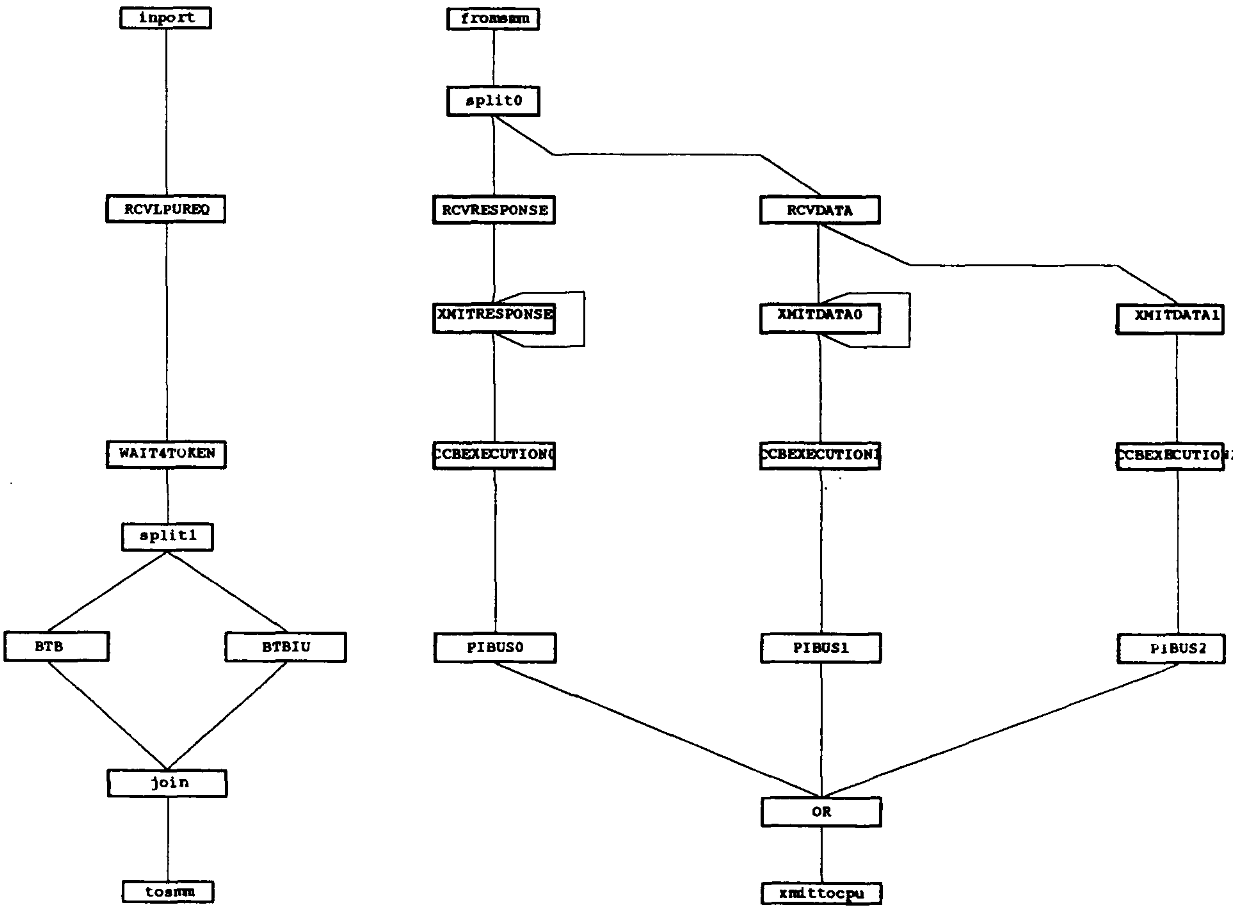

Figure A-25. BTB PASS on LPU to a CPU

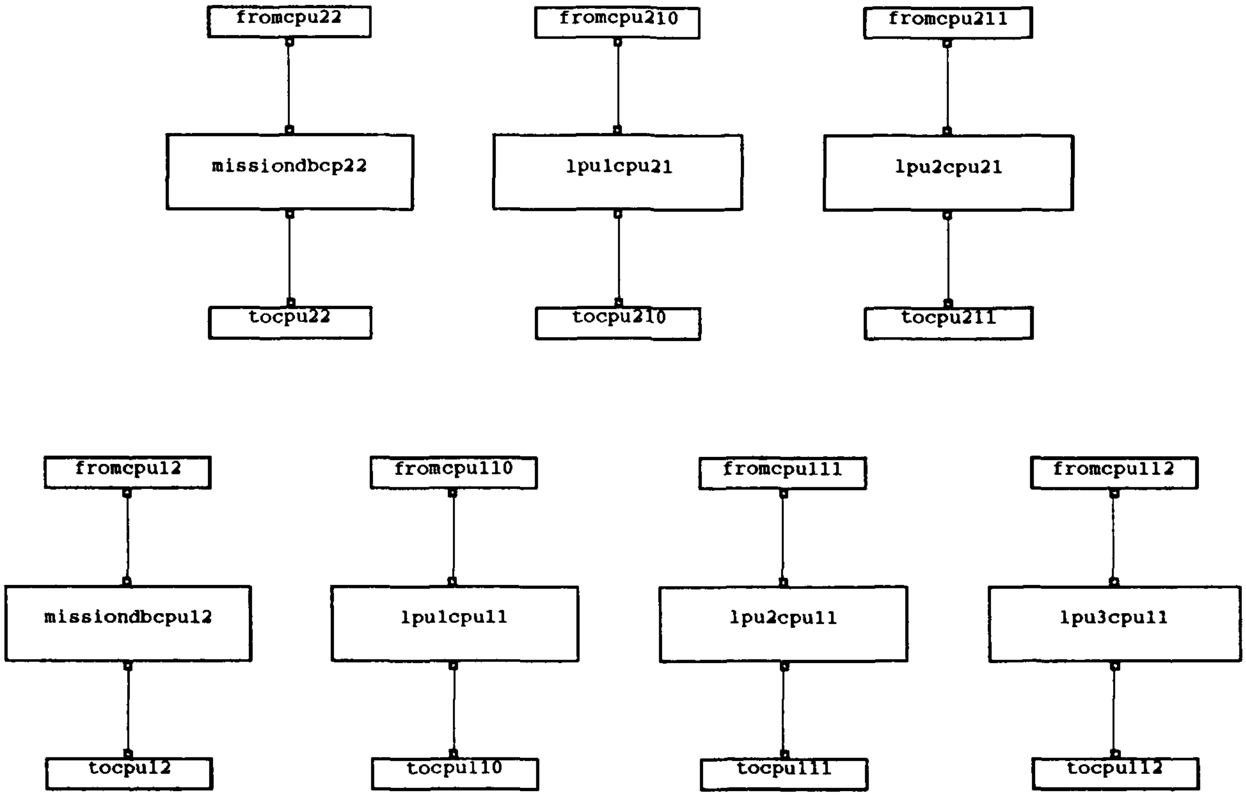

Figure A-26. SMM Download LPUs to CPUs

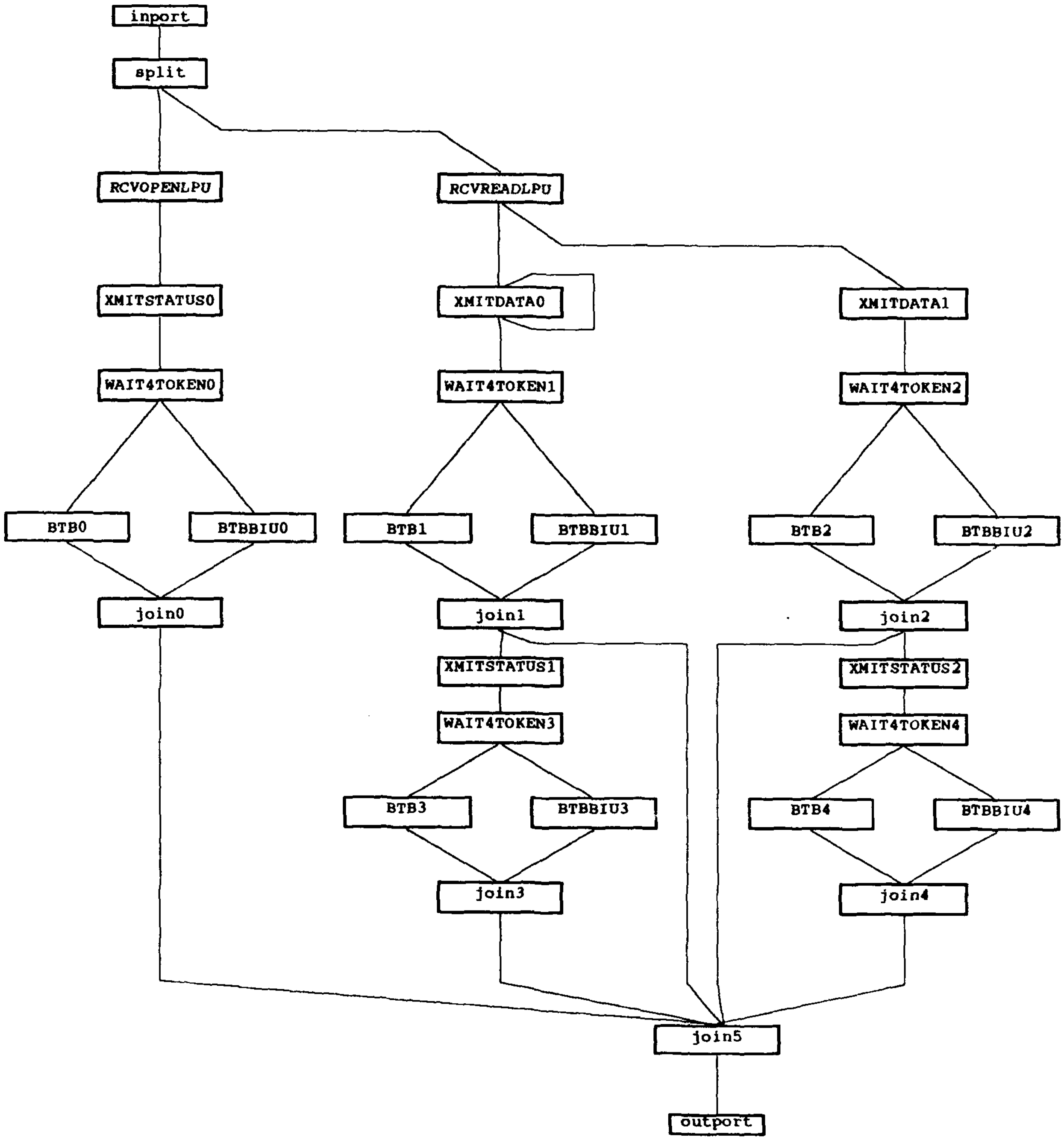

Figure A-27. SMM Download on LPU

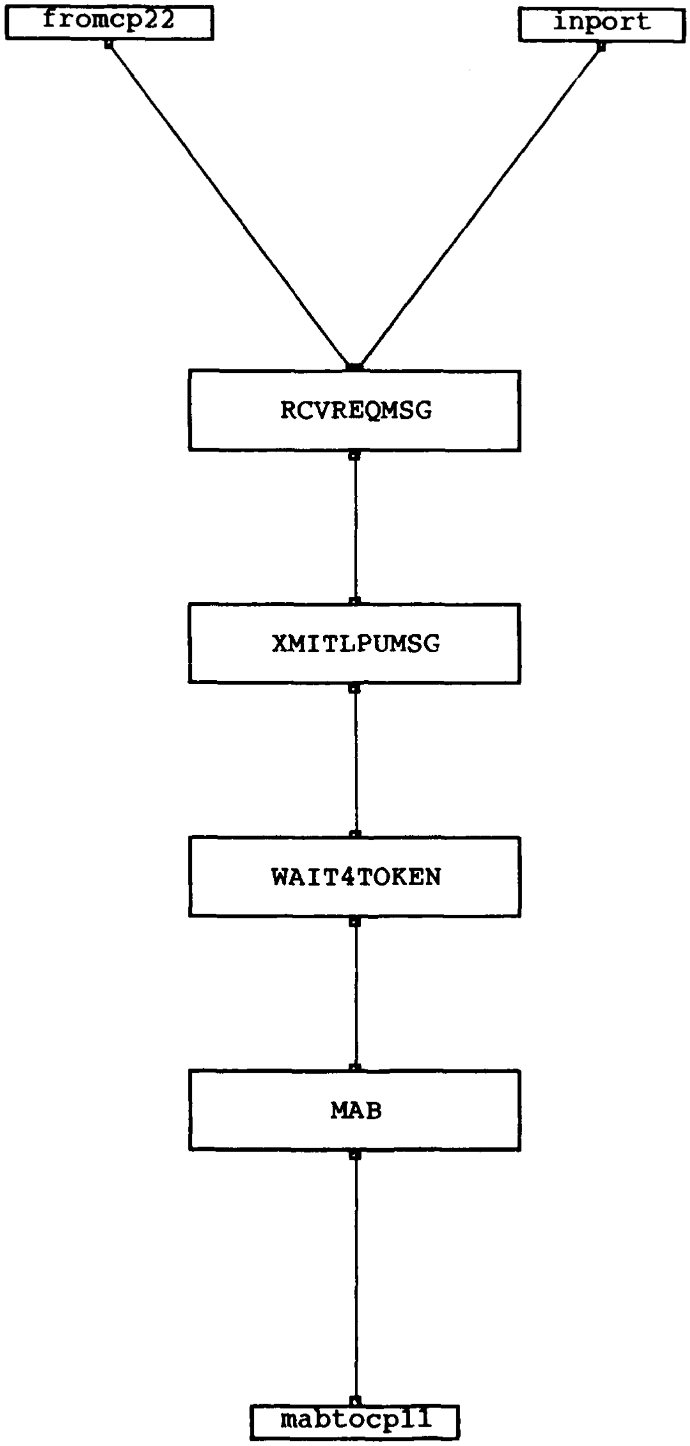

Figure A-28. Configuration Request Placed on MAB

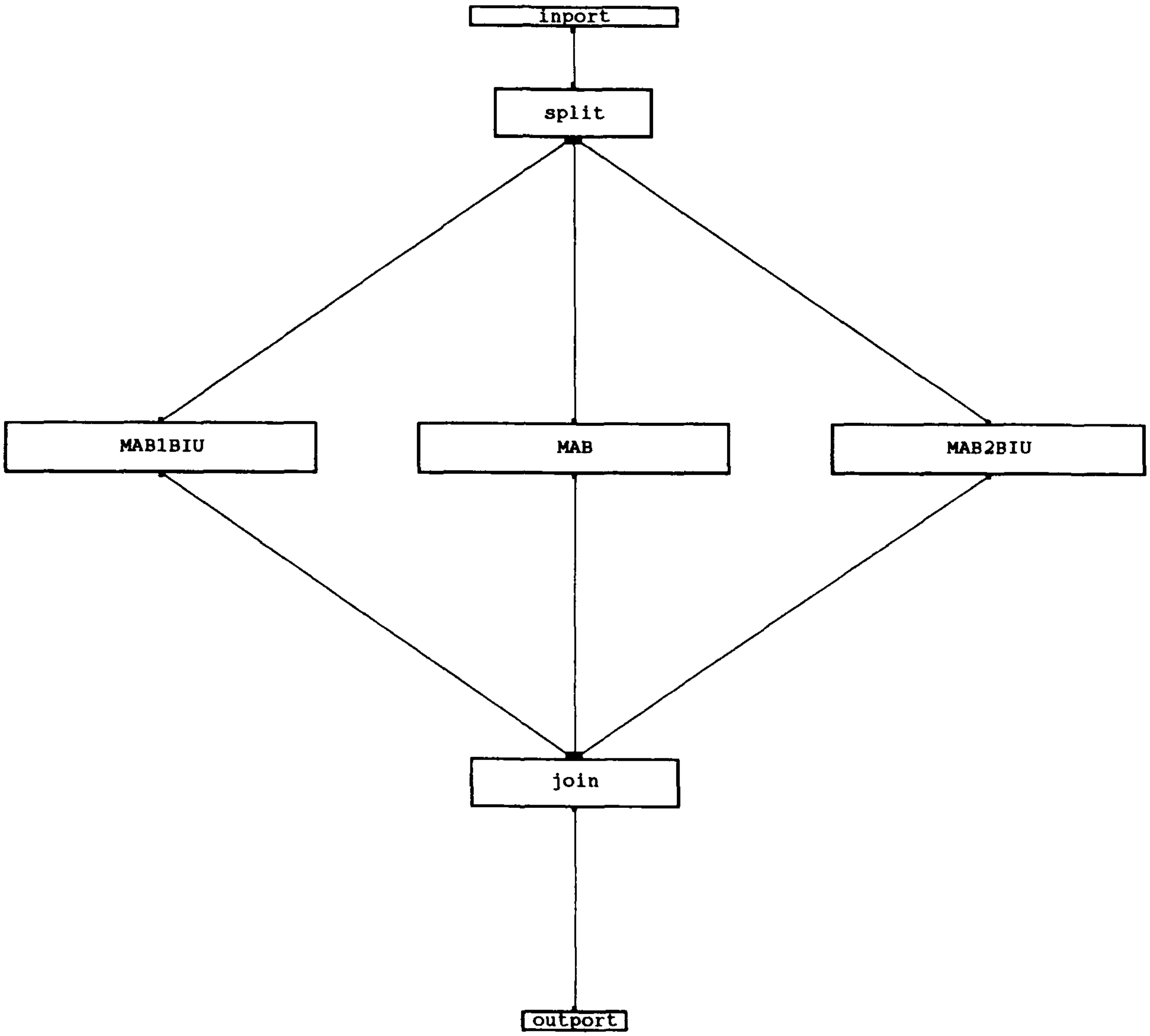

Figure A-29. MAB Transmission

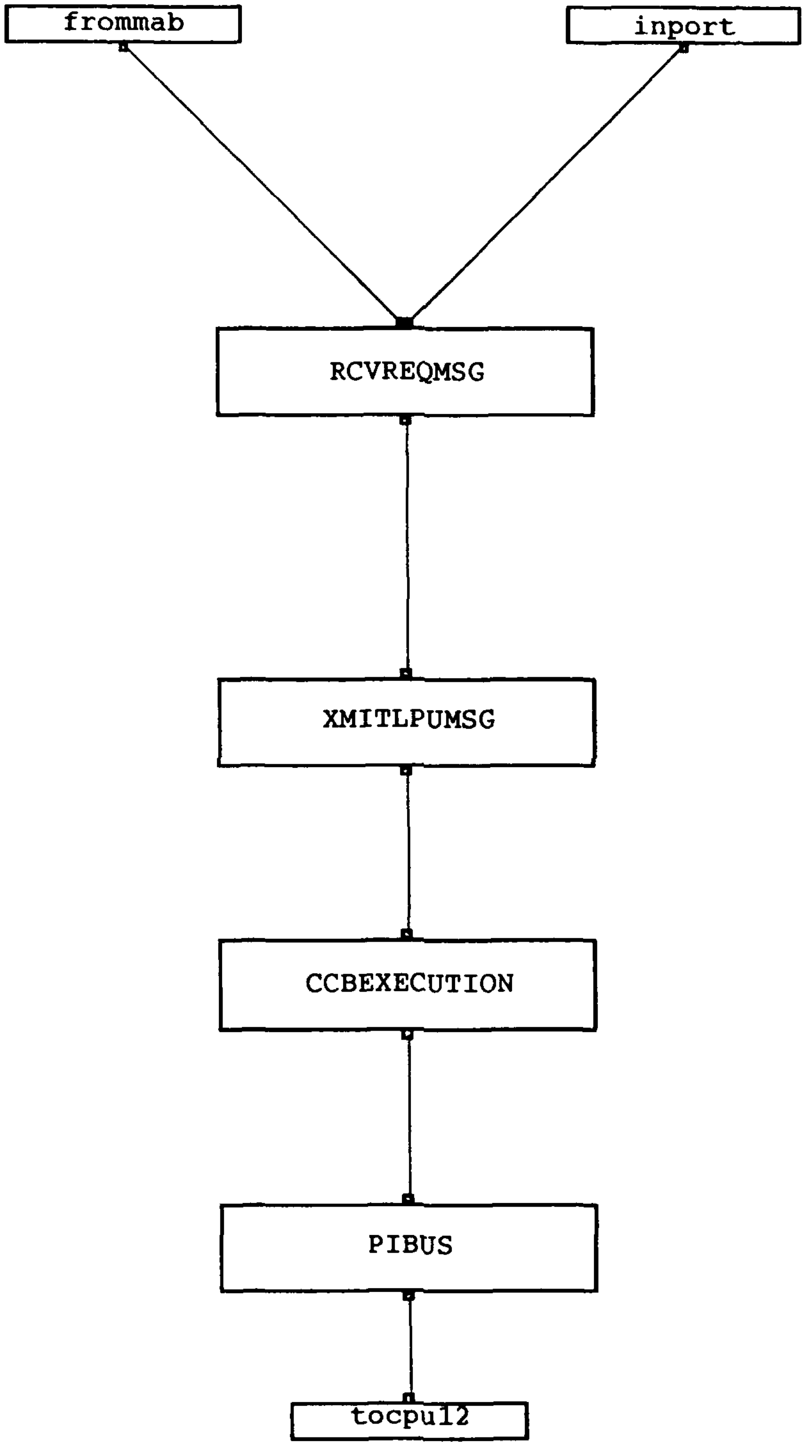

Figure A-30. MABIM Place Configuration Request on PI-bus

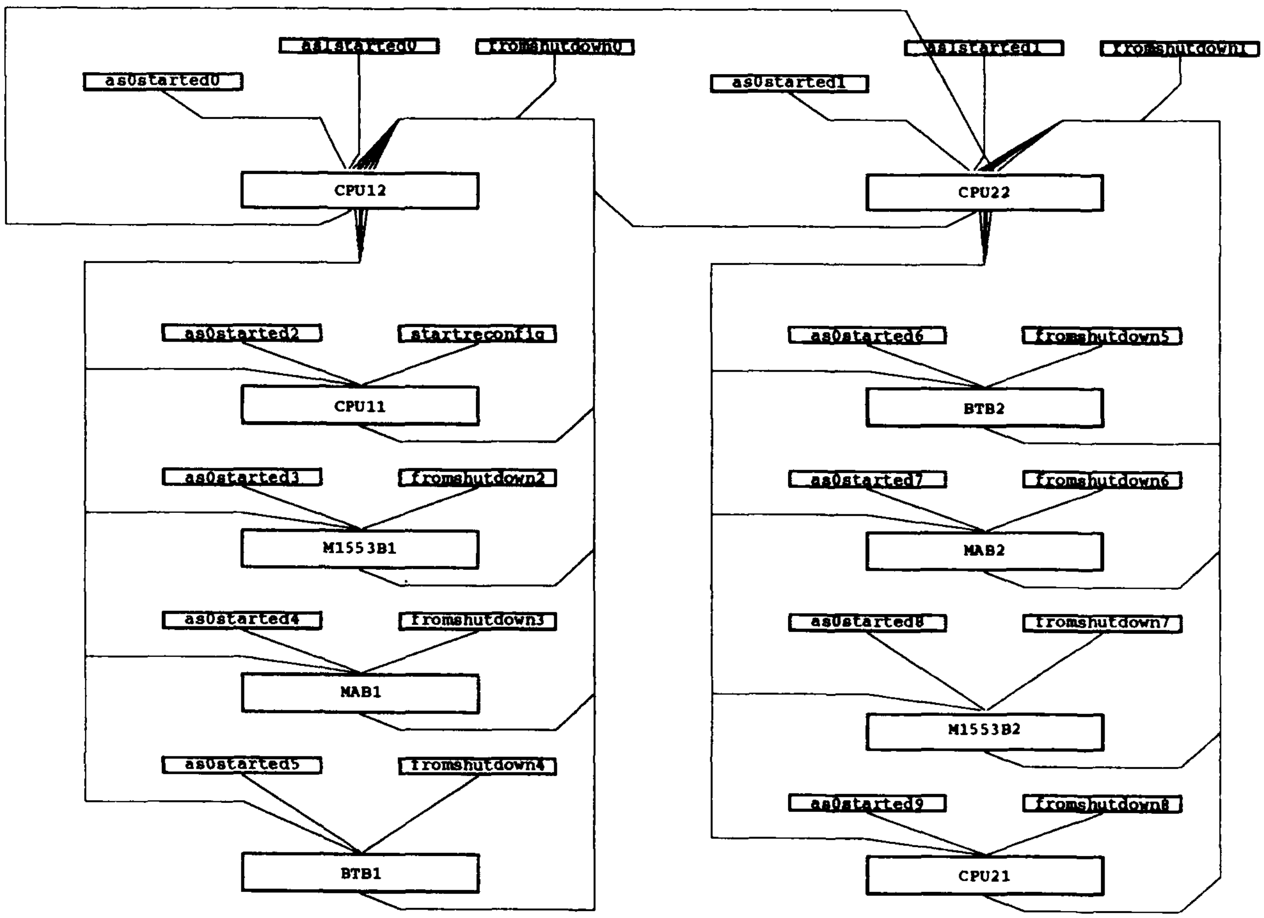

Figure A-31. System Messages Graph

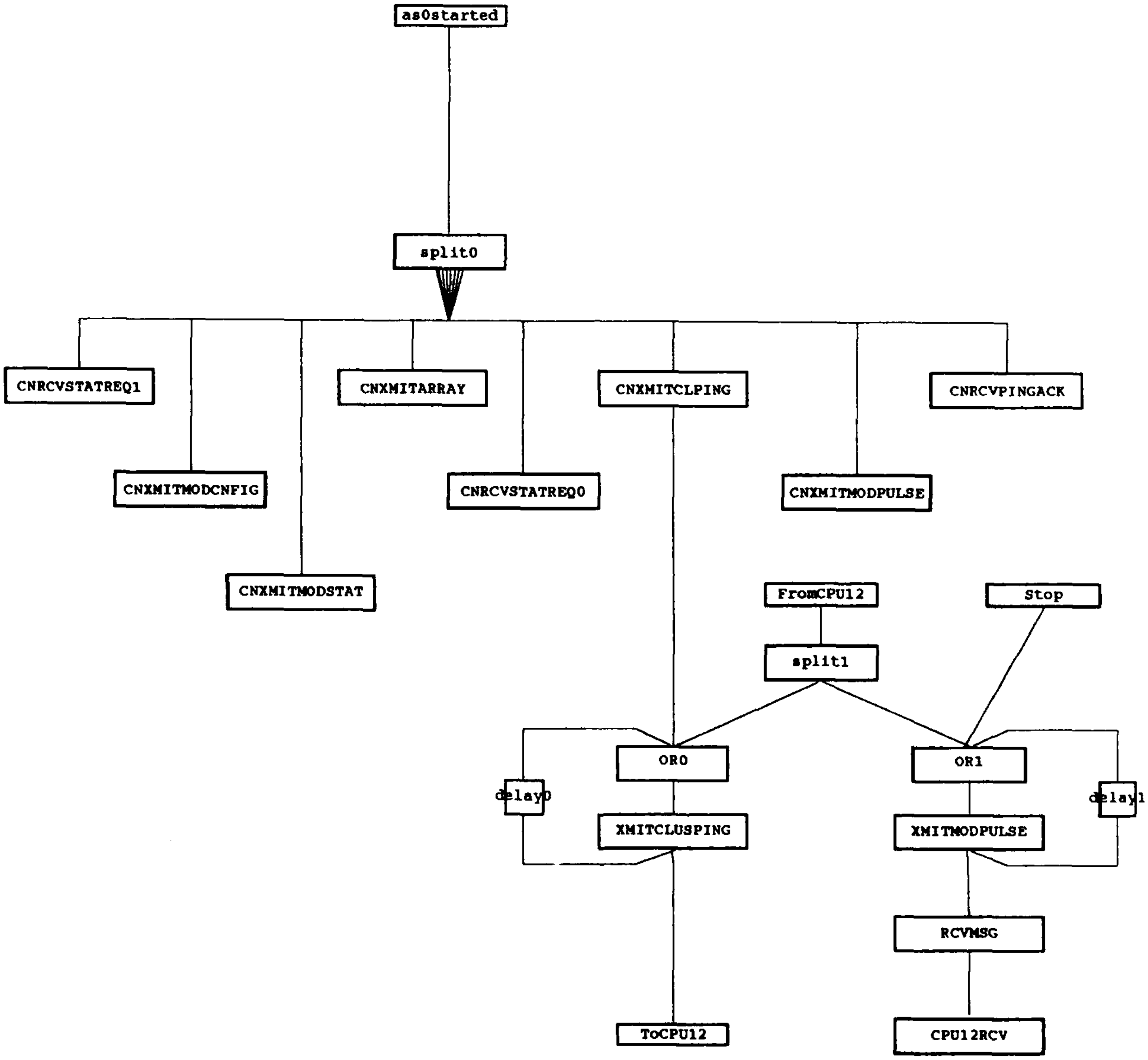

Figure A-32. Specialist System Messages

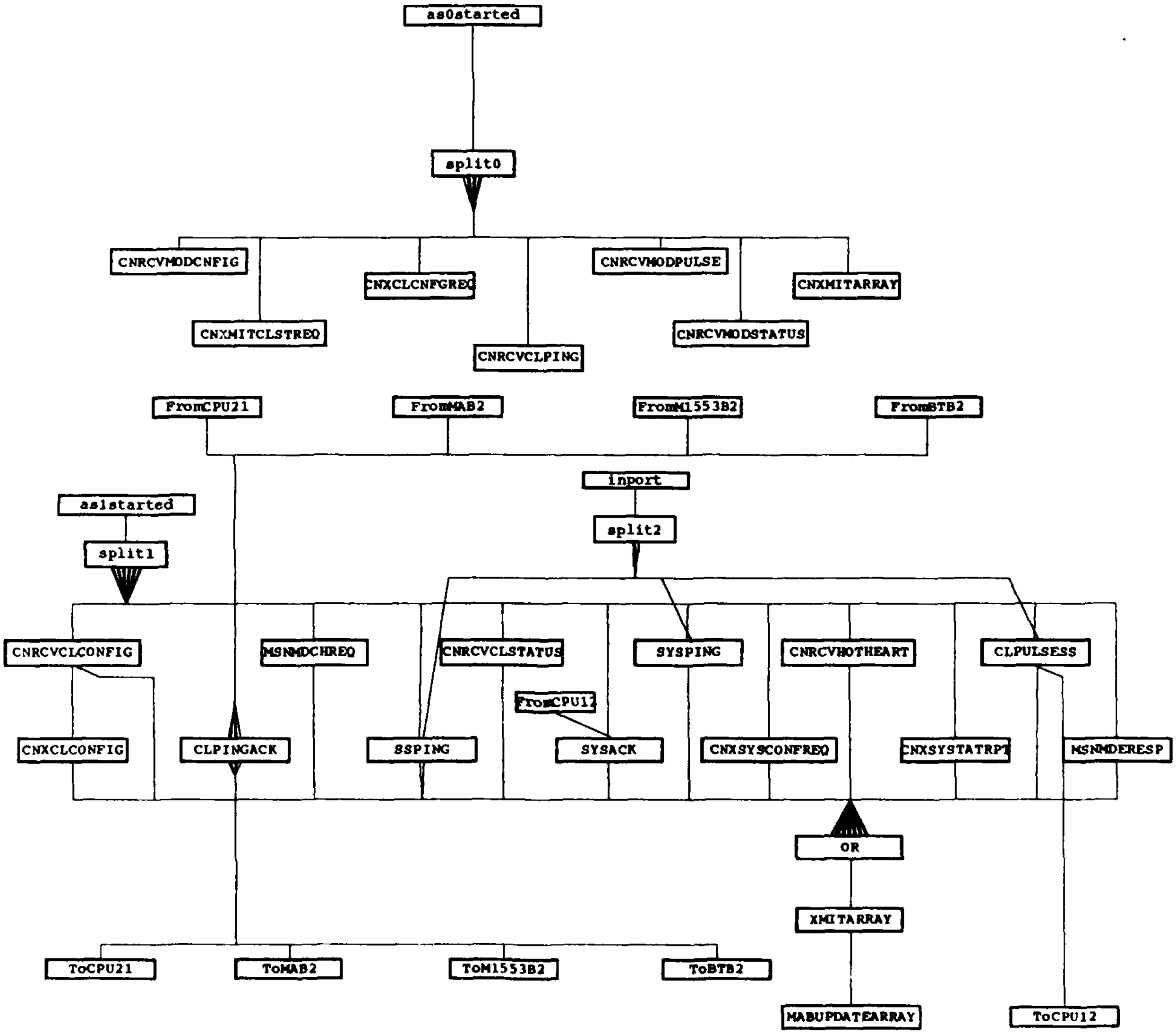

Figure A-33. Supervisor Module System Messages

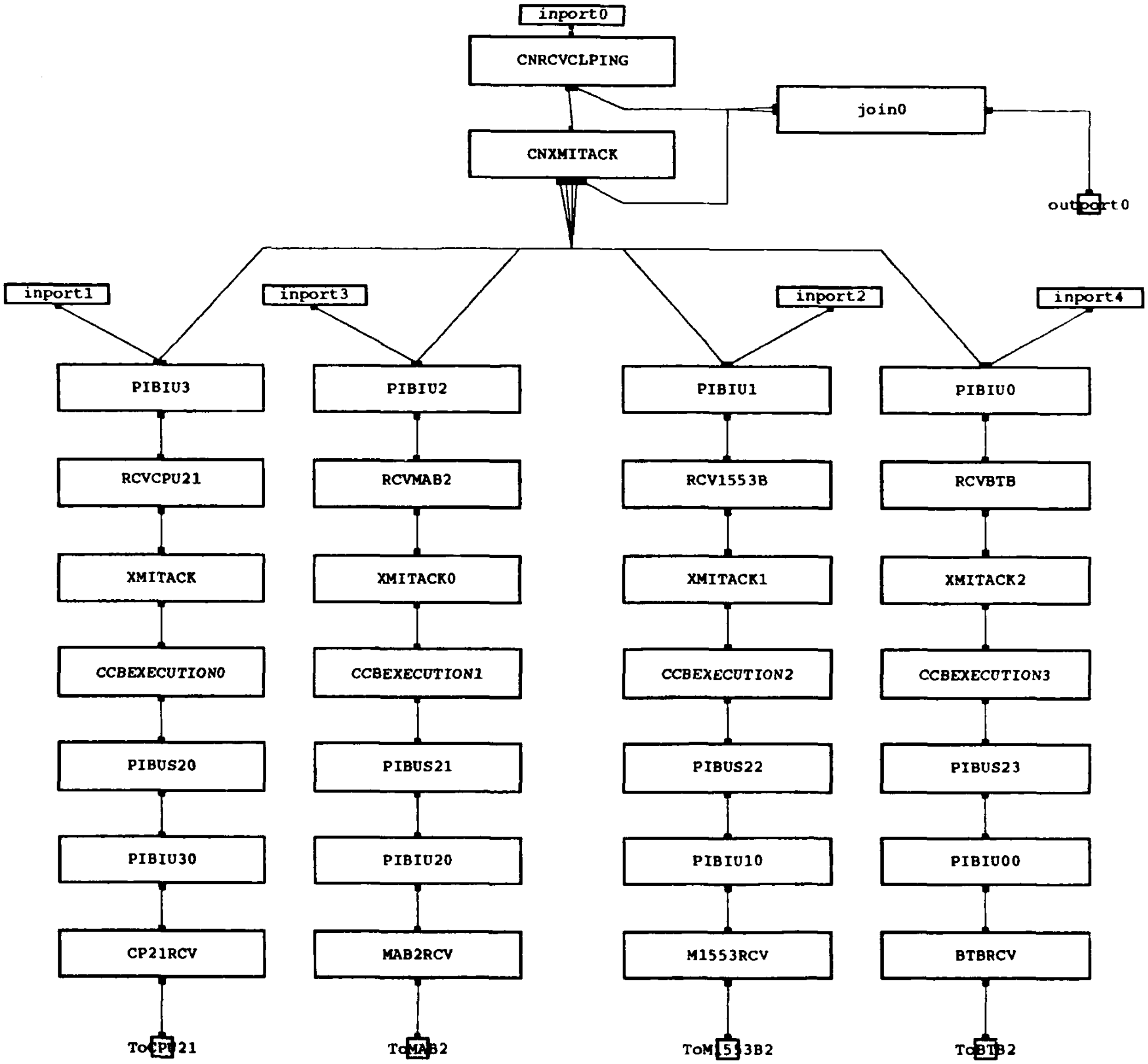

Figure A-34. Cluster Supervisor Acknowledge Ping

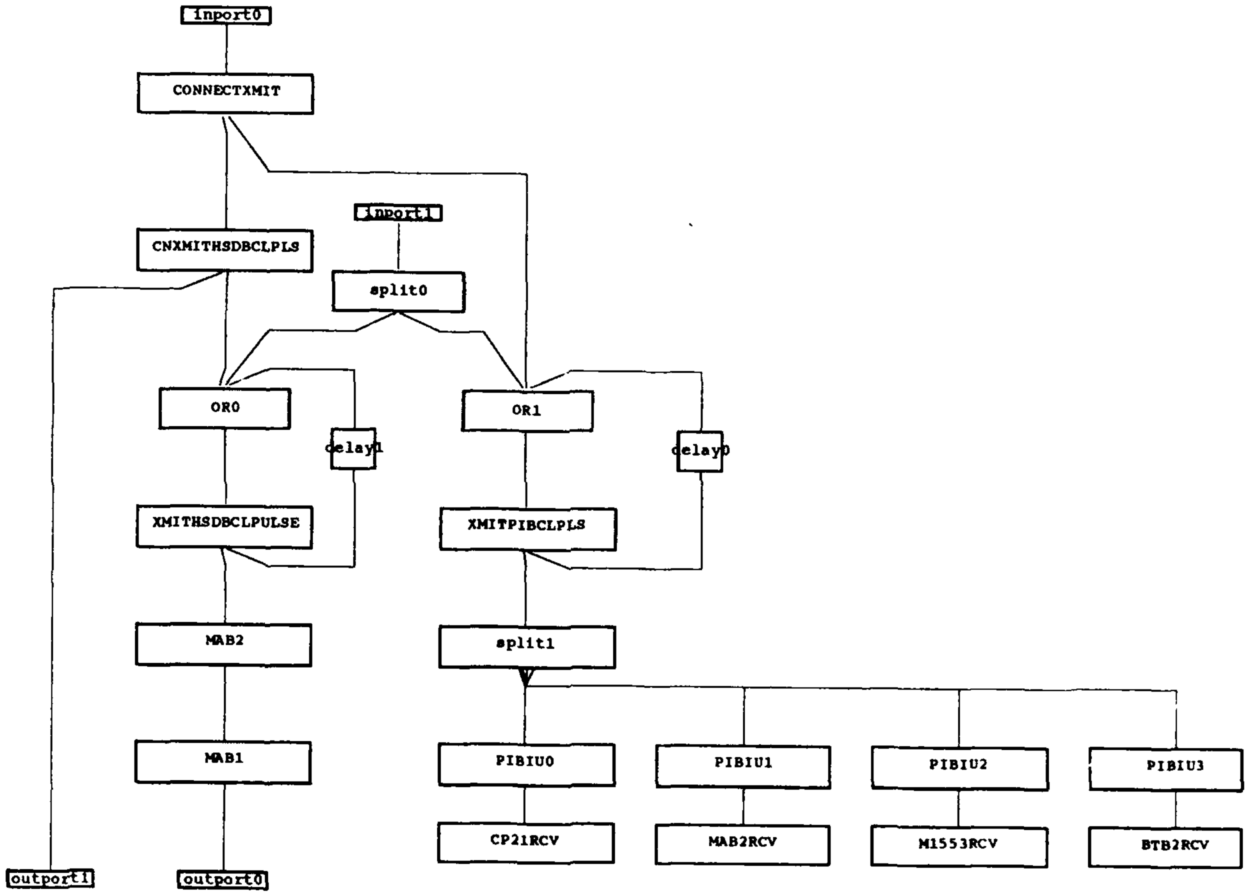

Figure A-35. System Supervisor Heartbeats

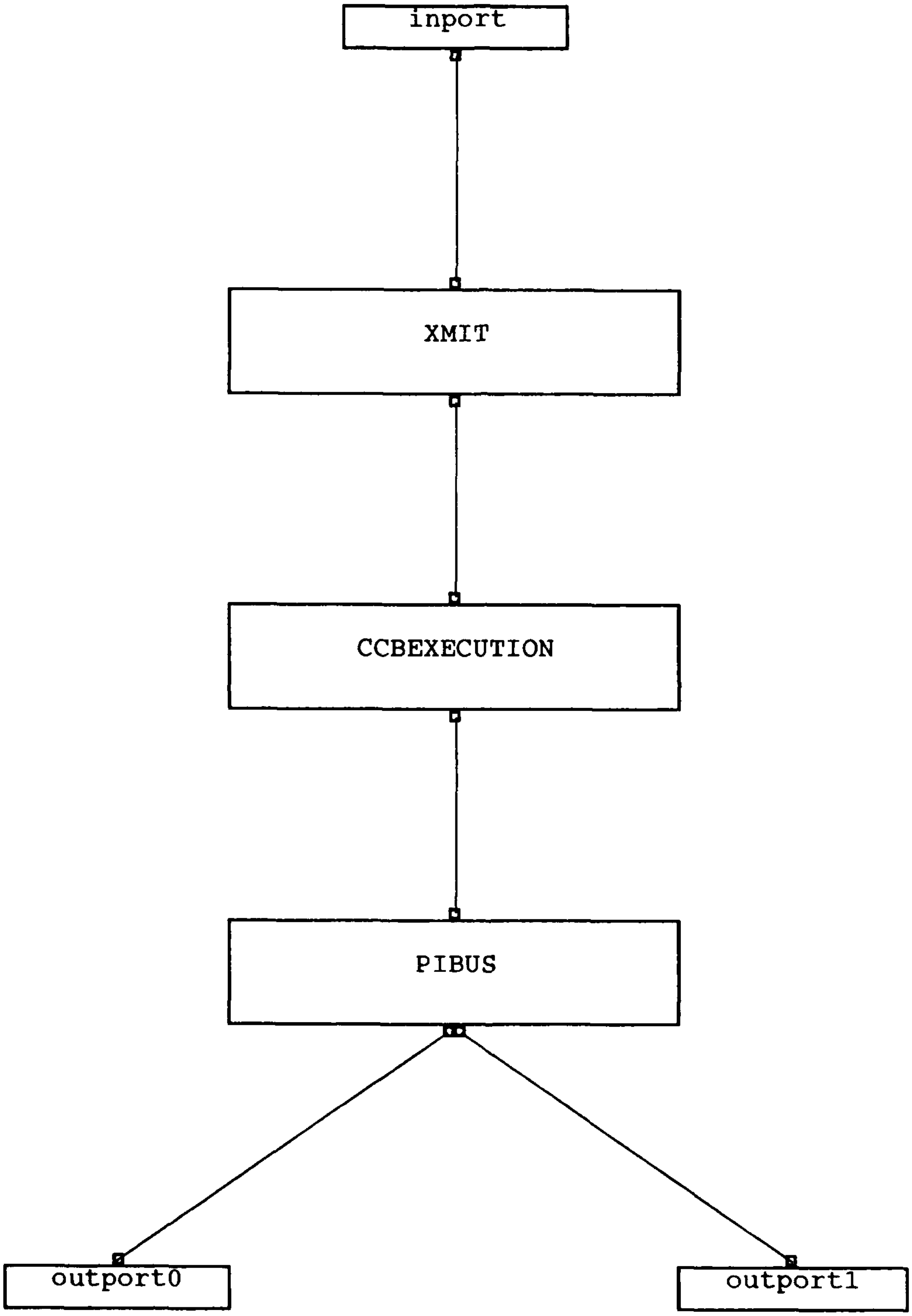

Figure A-36. Transmit Ping or Pulse

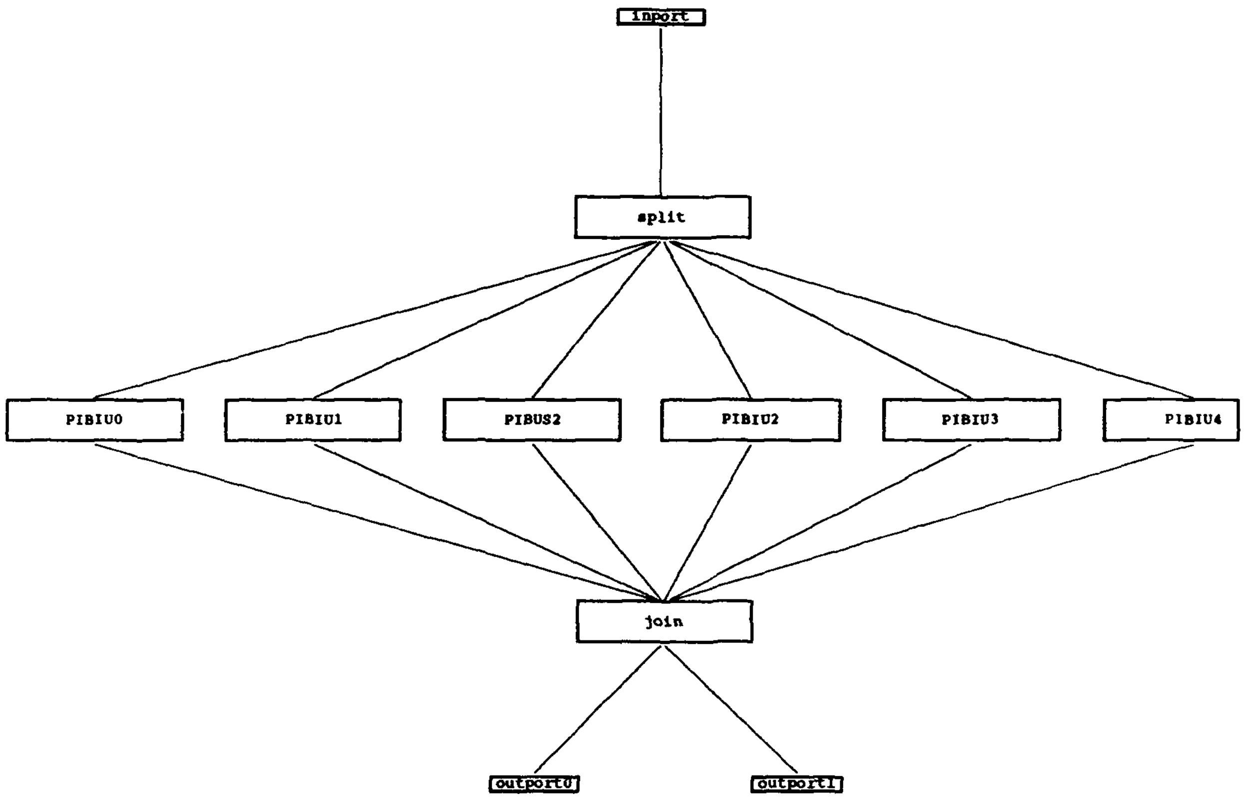

Figure A-37. PI-bus Broadcast

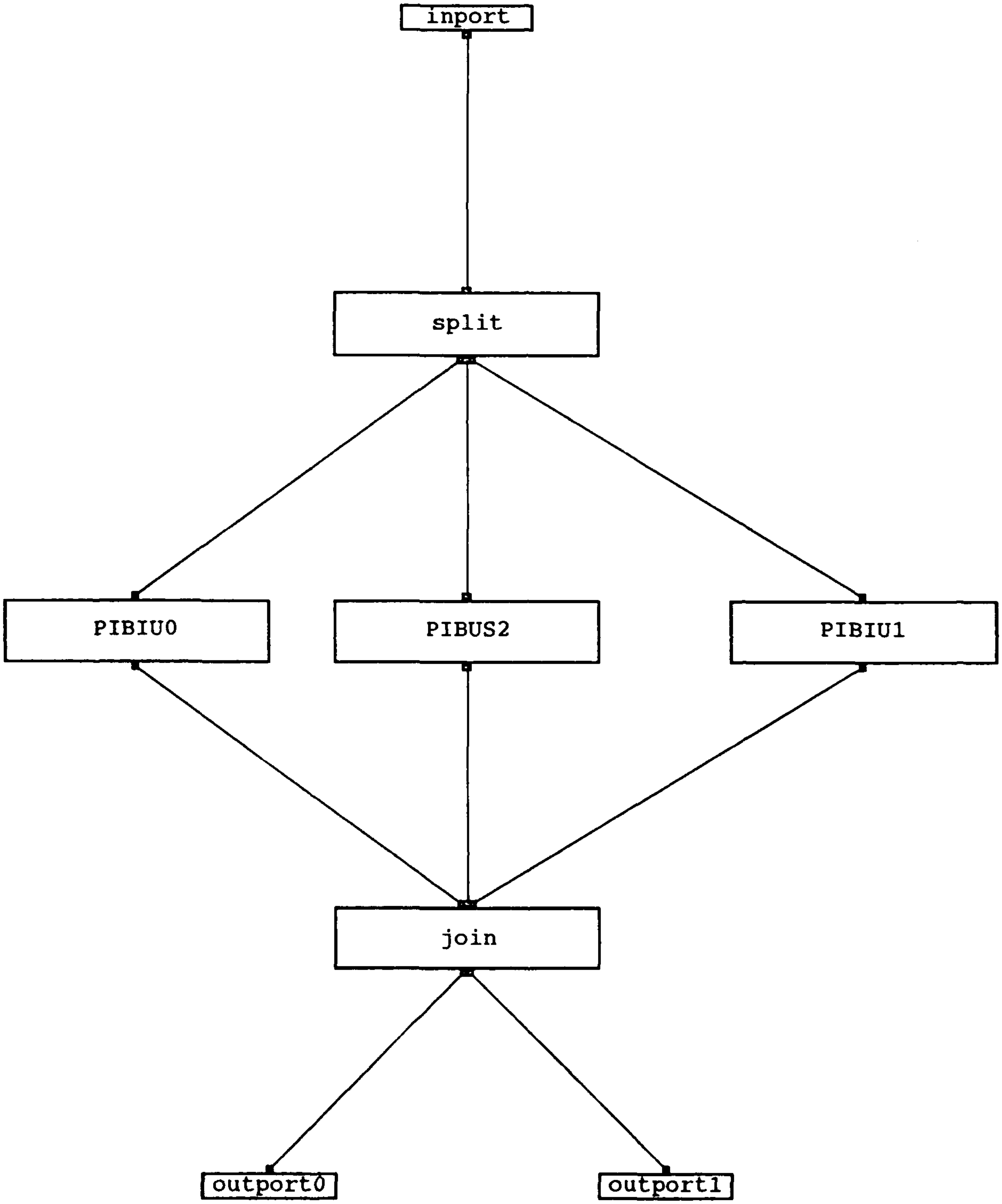

Figure A-38. PI-bus Transmission with Two Outputs

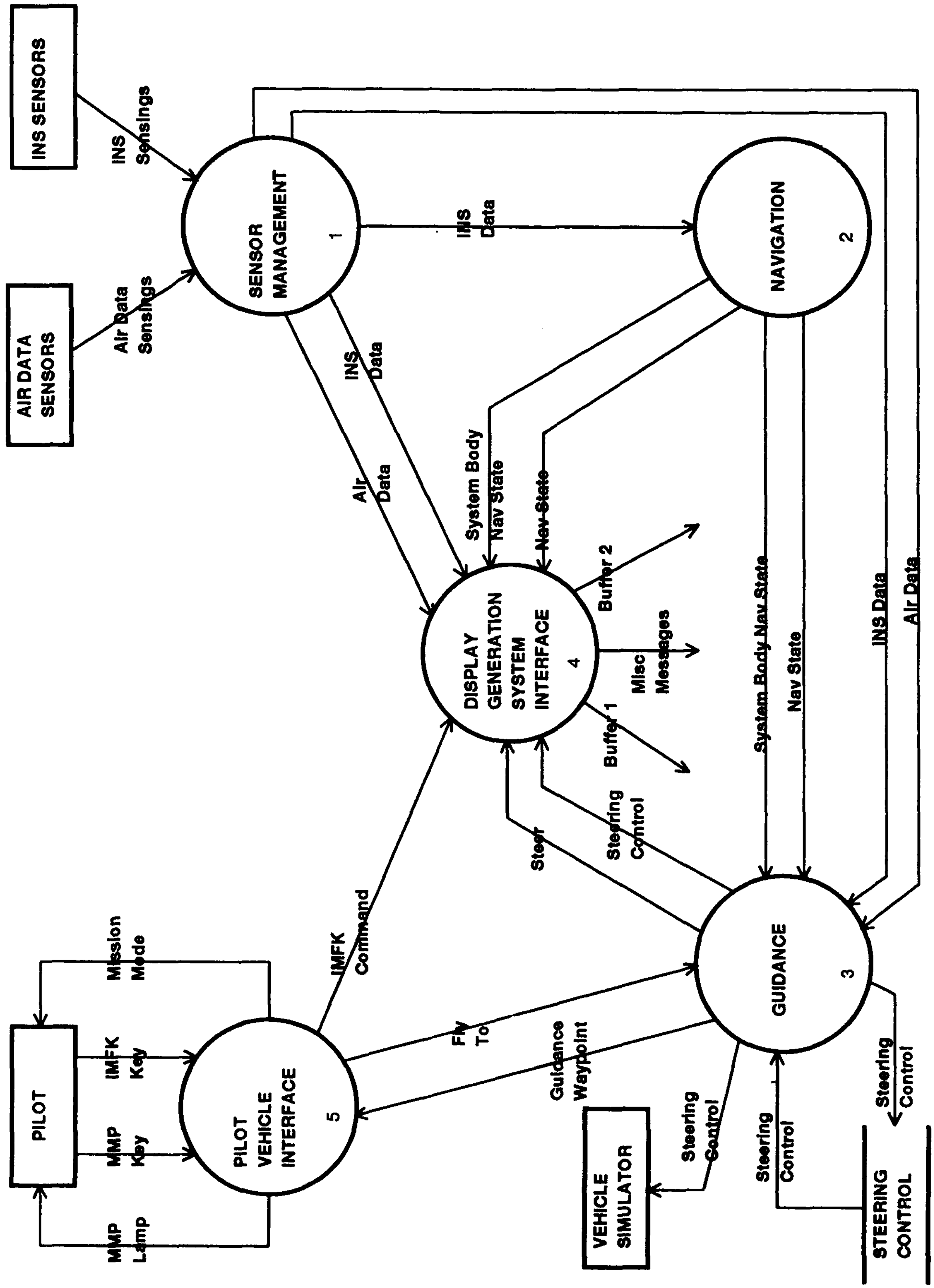

Figure A-39. Dataflow of Demonstration 3

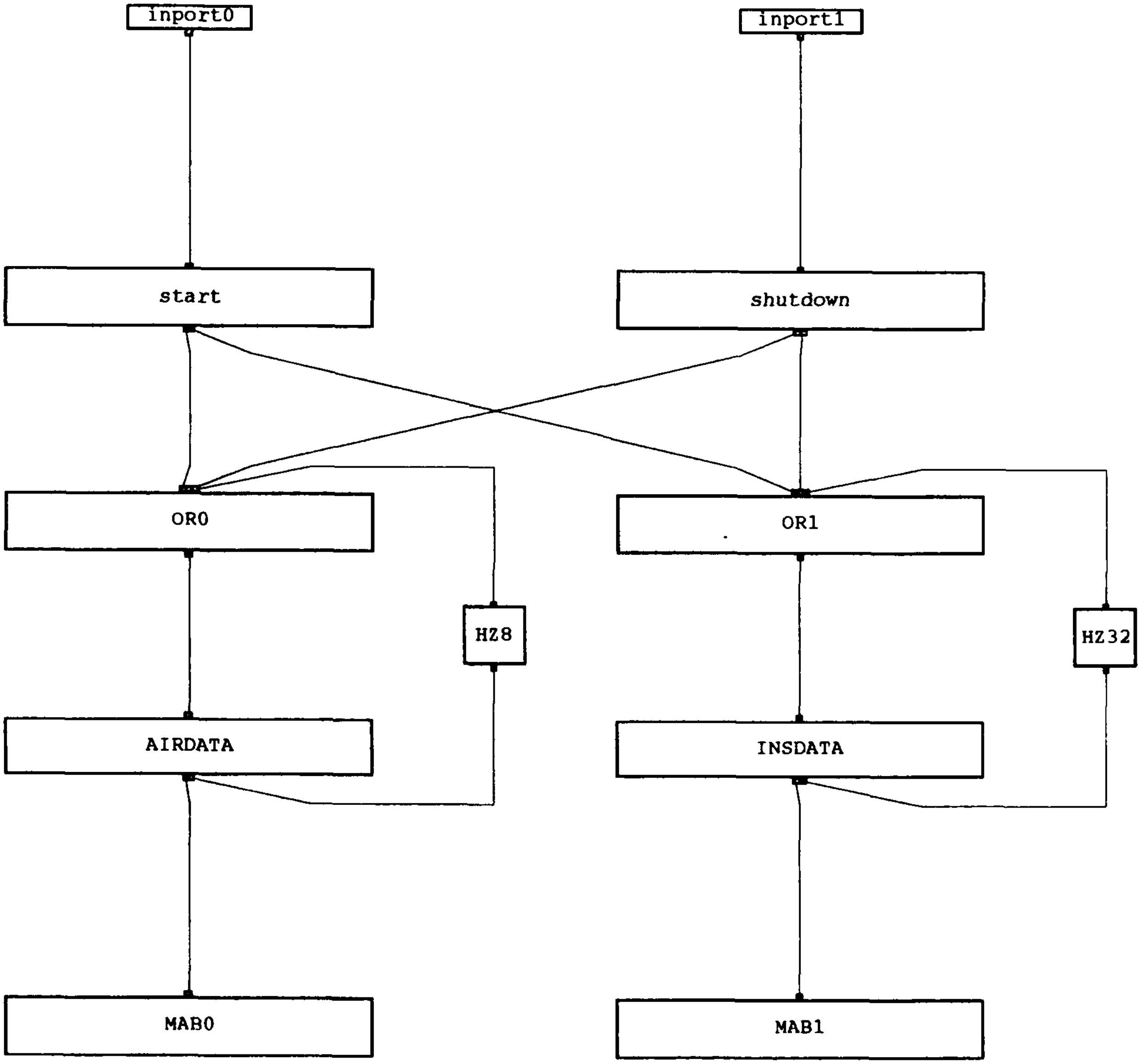

Figure A-40. Sensor Input

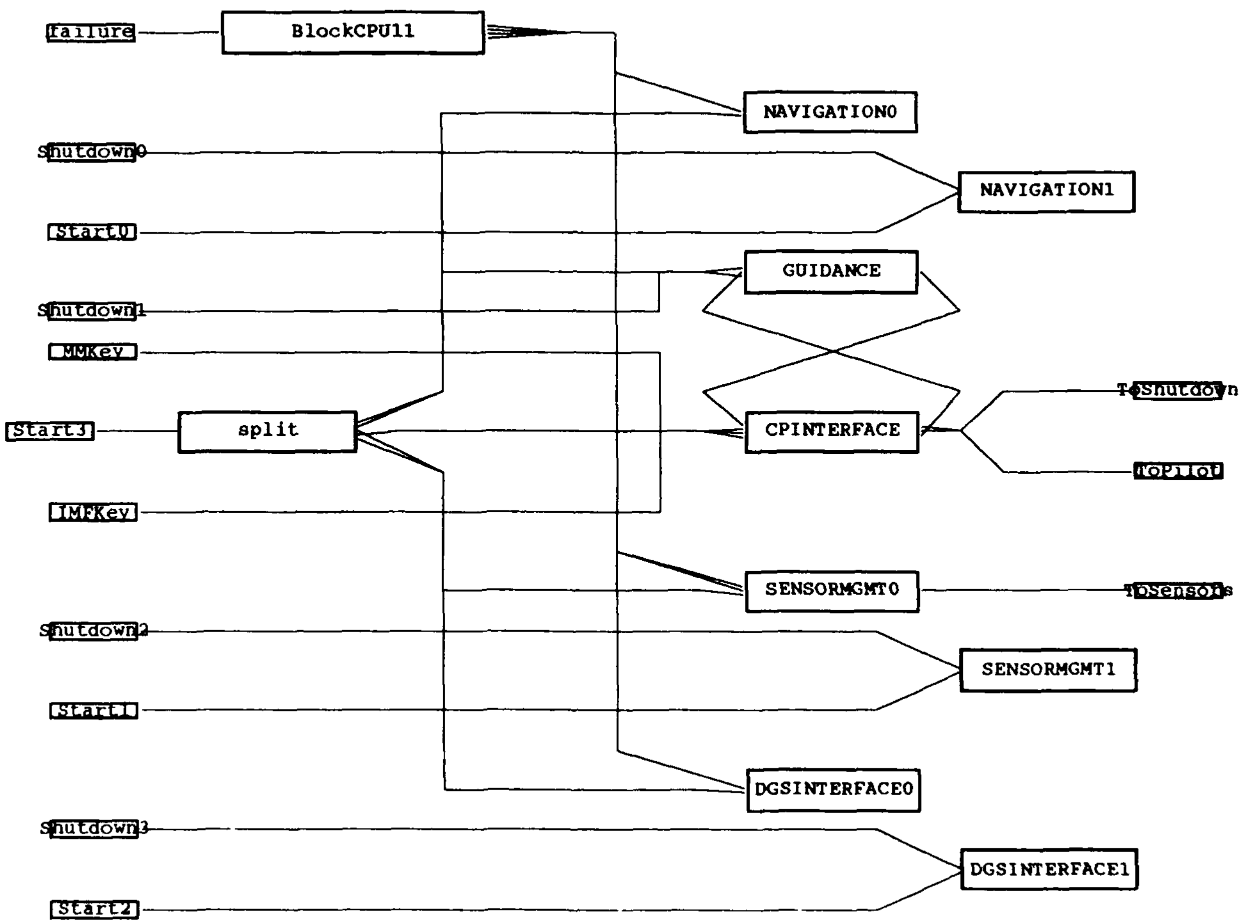

Figure A-41. Normal Operations

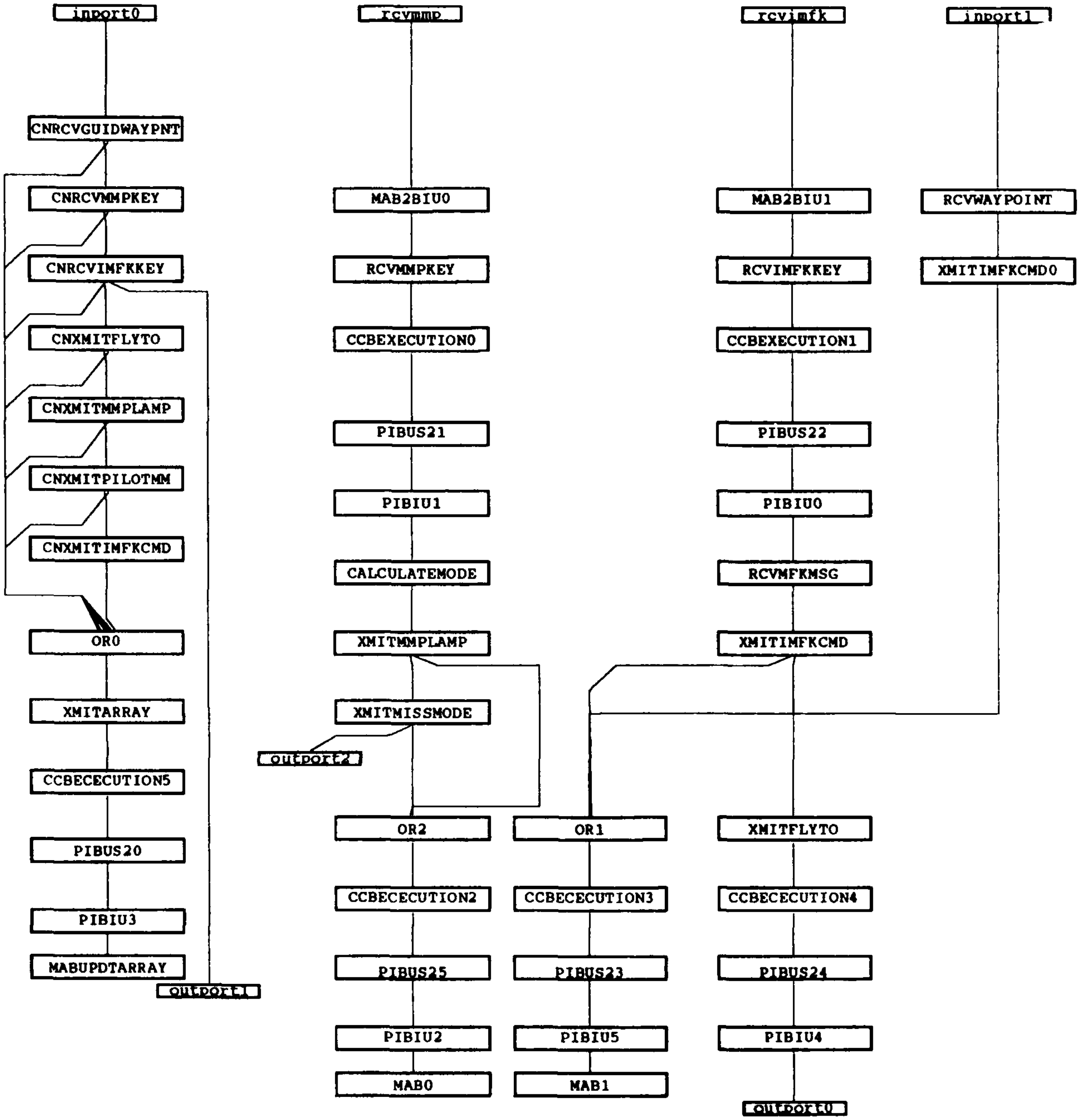

Figure A-42. Cockpit Interface LPU in CPU21

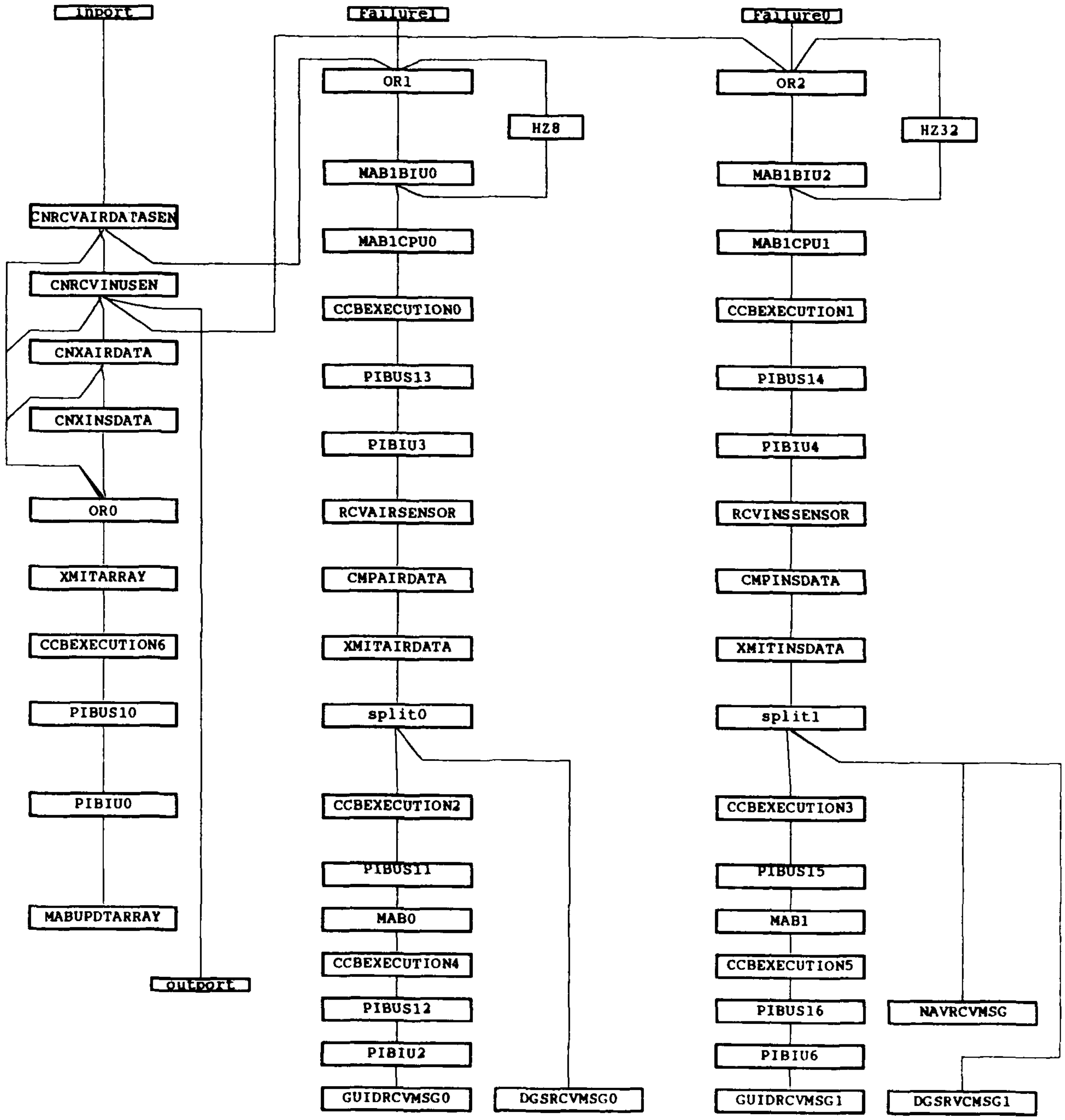

Figure A-43. Sensor Management LPU in CPU11

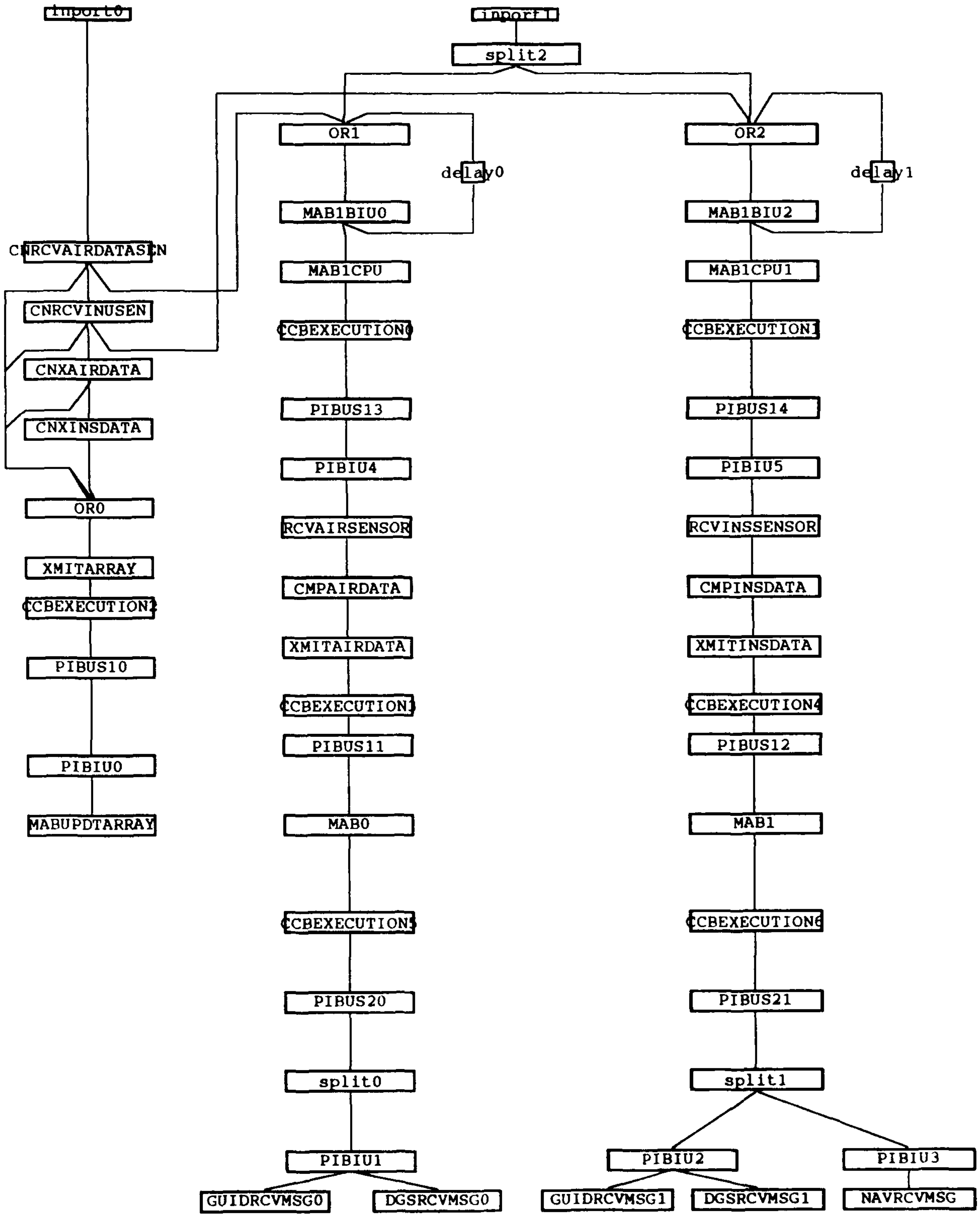

Figure A-44. Sensor Management LPU in CPU12

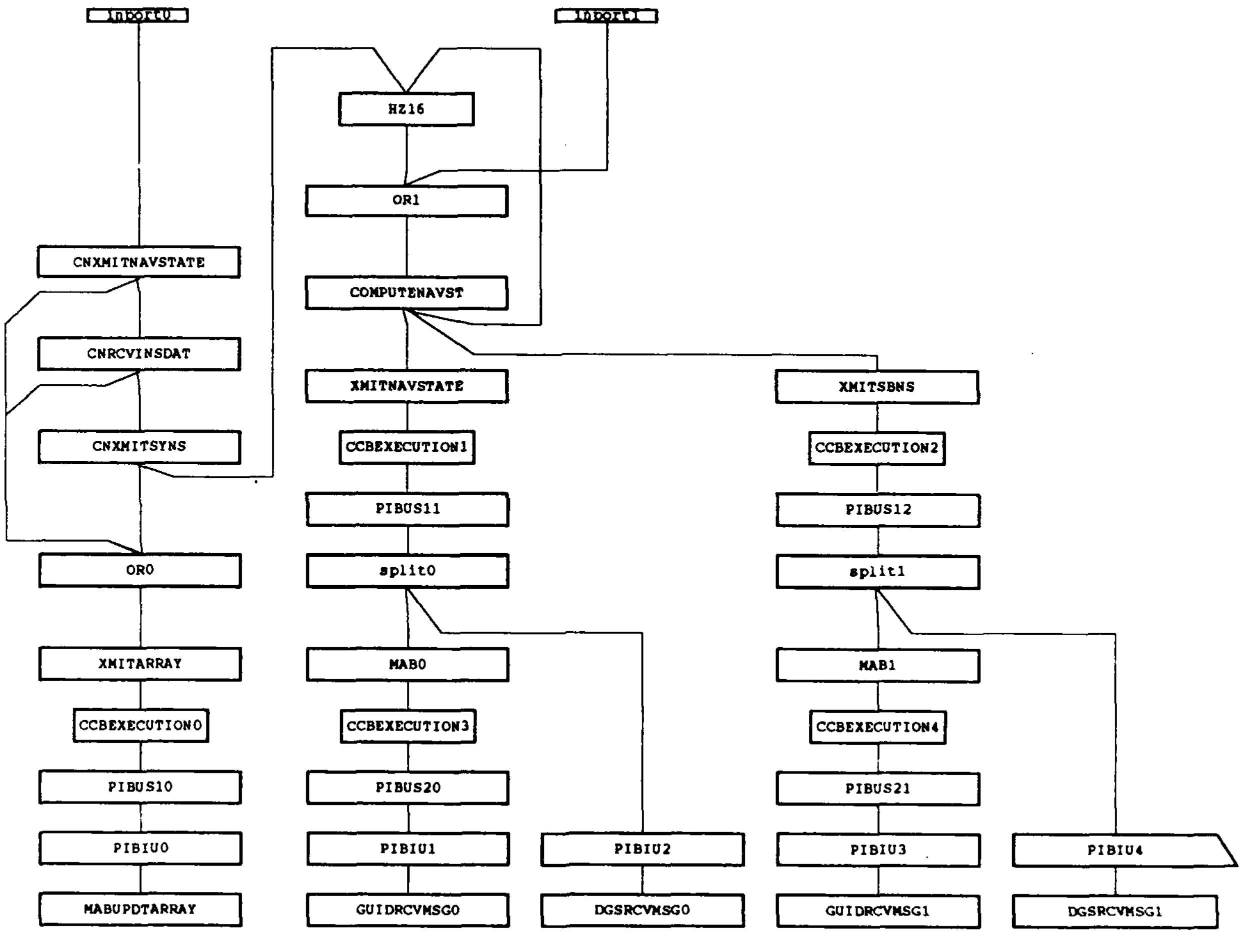

Figure A-45. Navigation LPU in CPU11

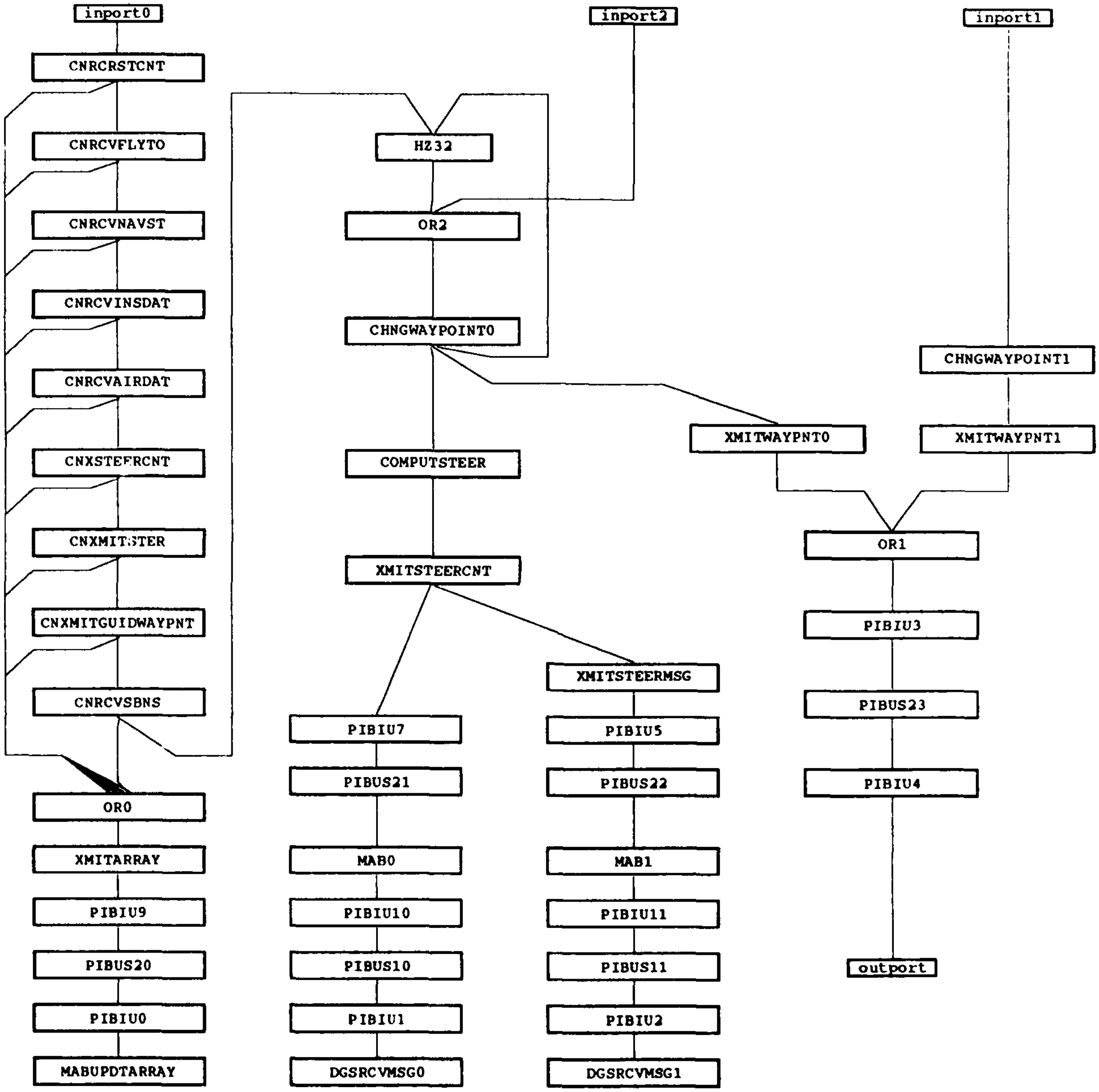

Figure A-46. Guidance LPU in CPU21

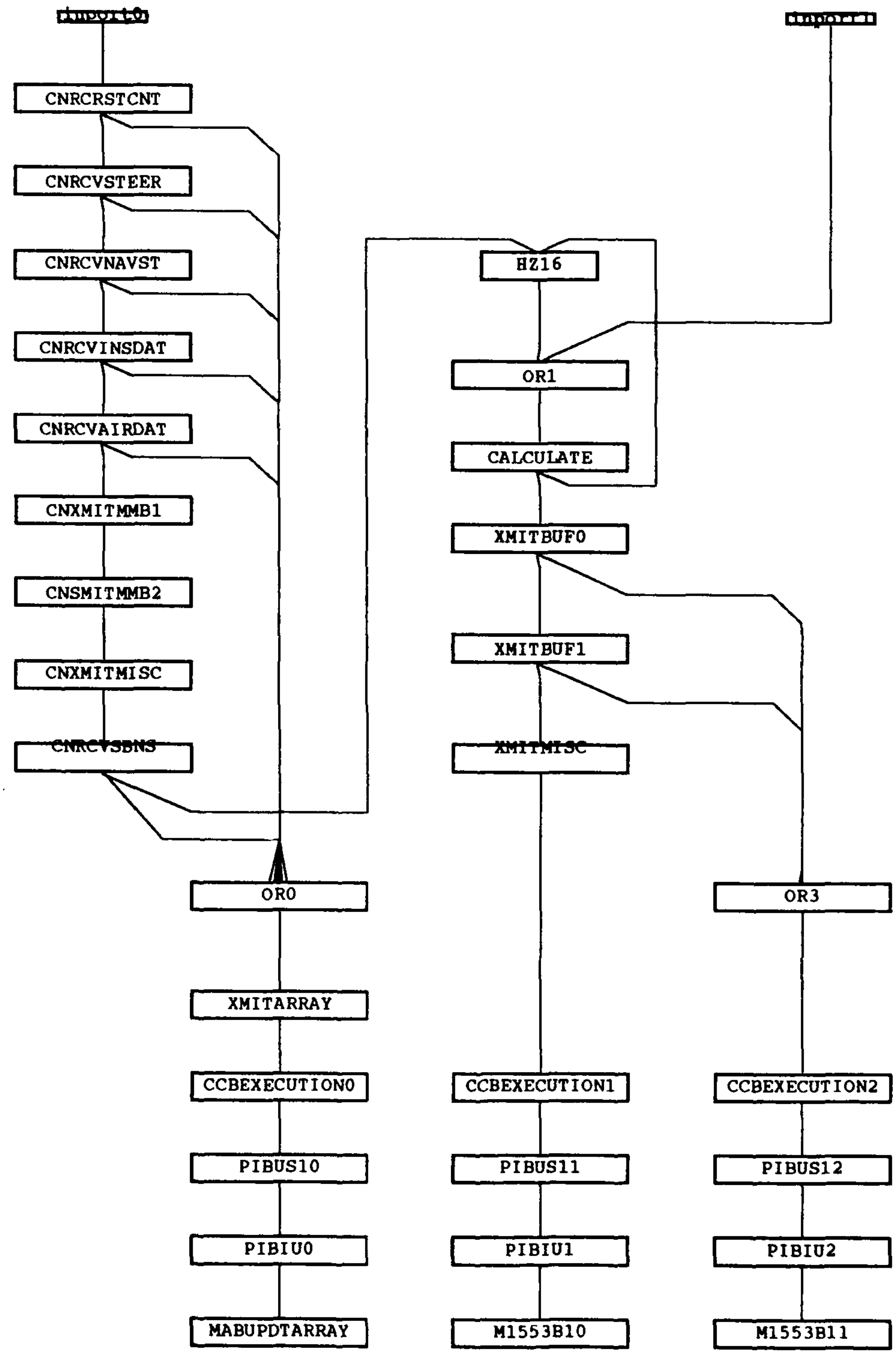

Figure A-47. DGS Interface LPU in CPU11

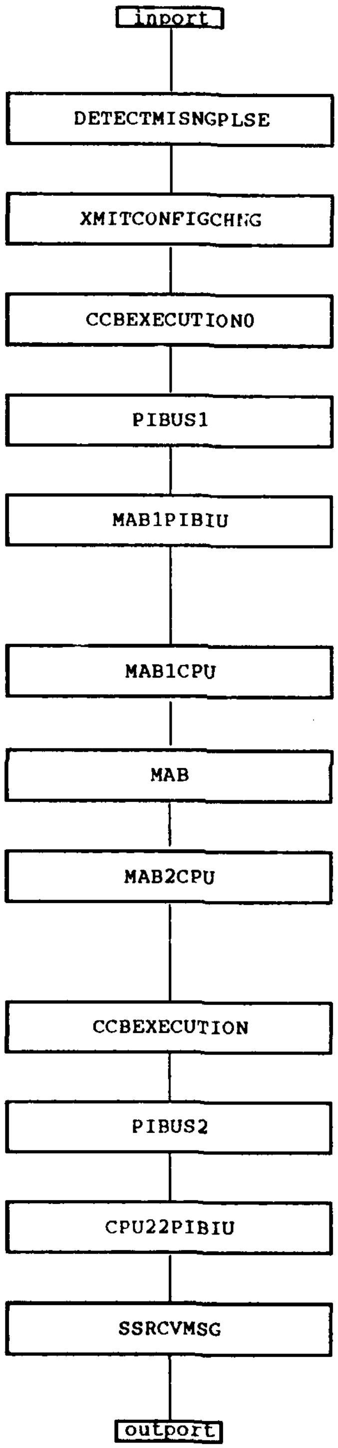

Figure A-48. Detection and Reporting of Failed CPU

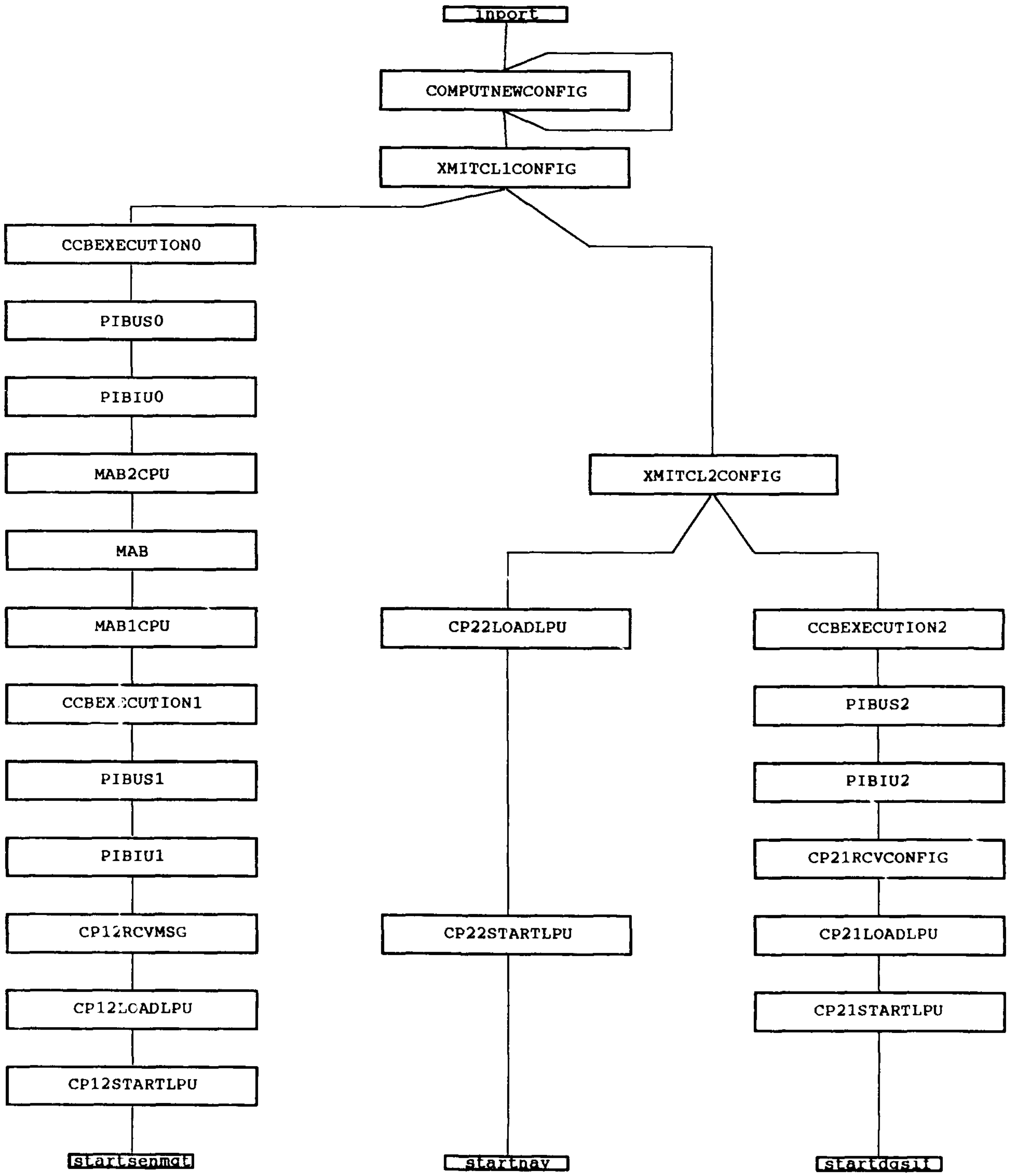

Figure A-49. Reconfiguration

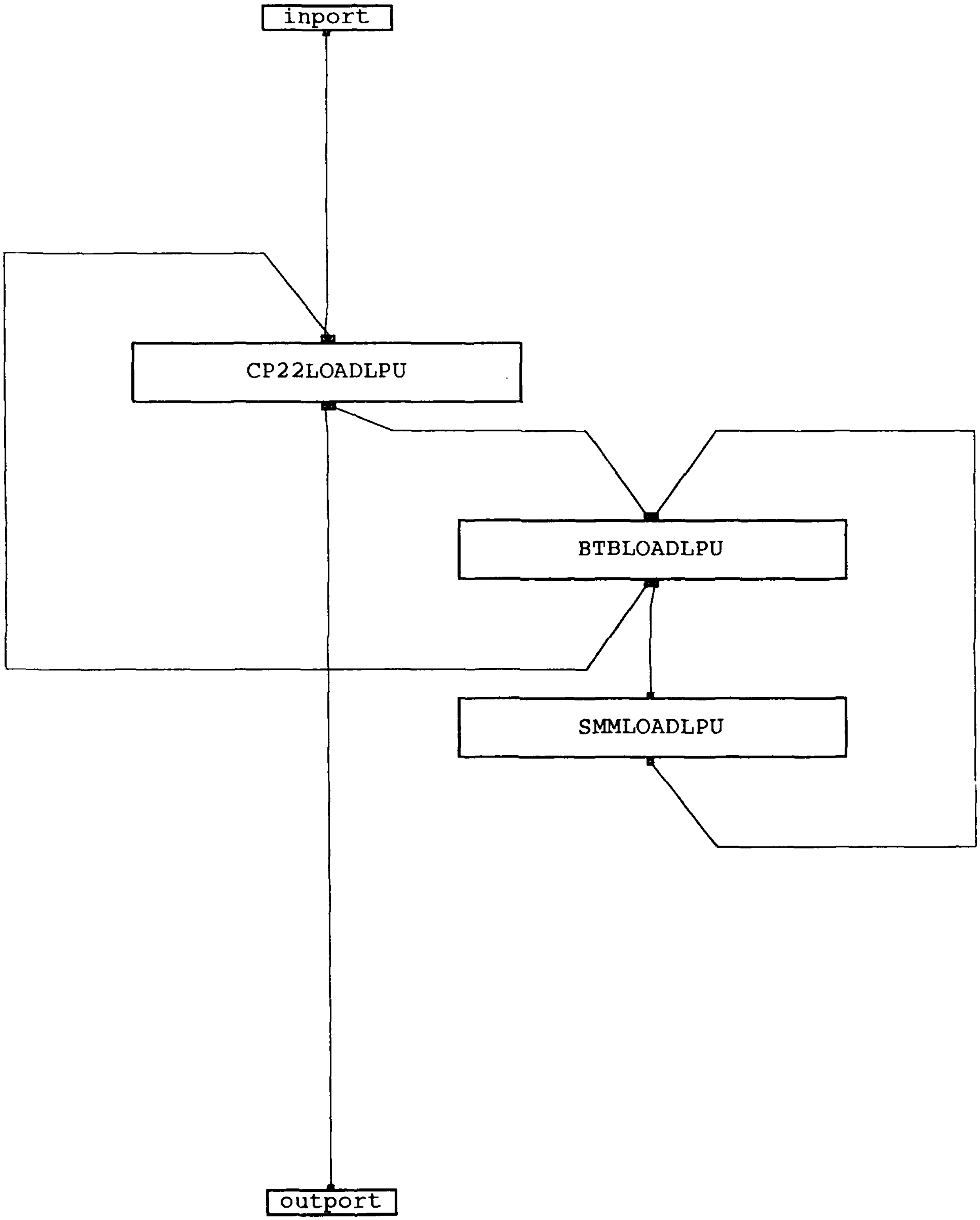

Figure A-50. Load on LPU

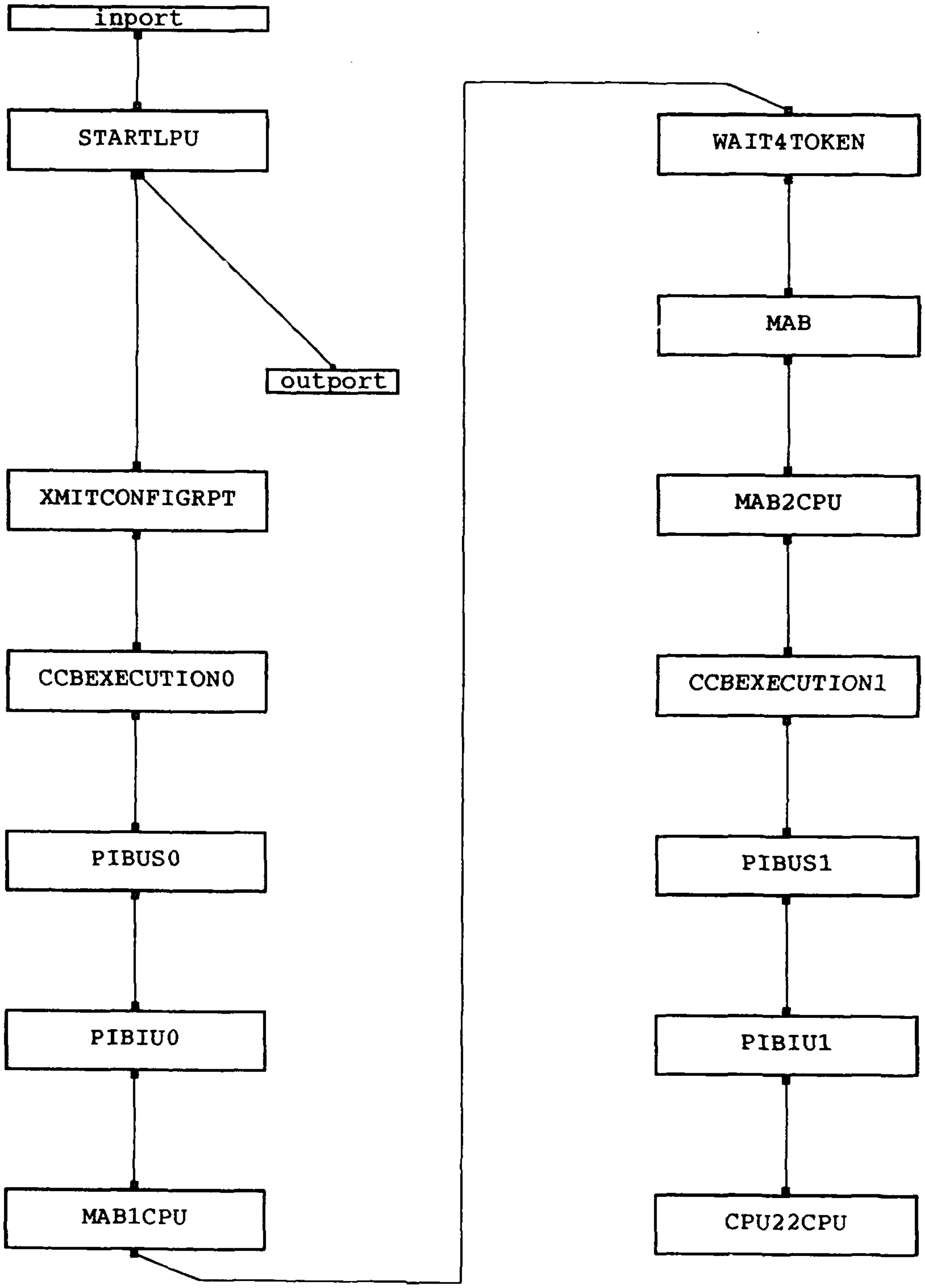

Figure A-51. Start on LPU and Transmit Configuration Report

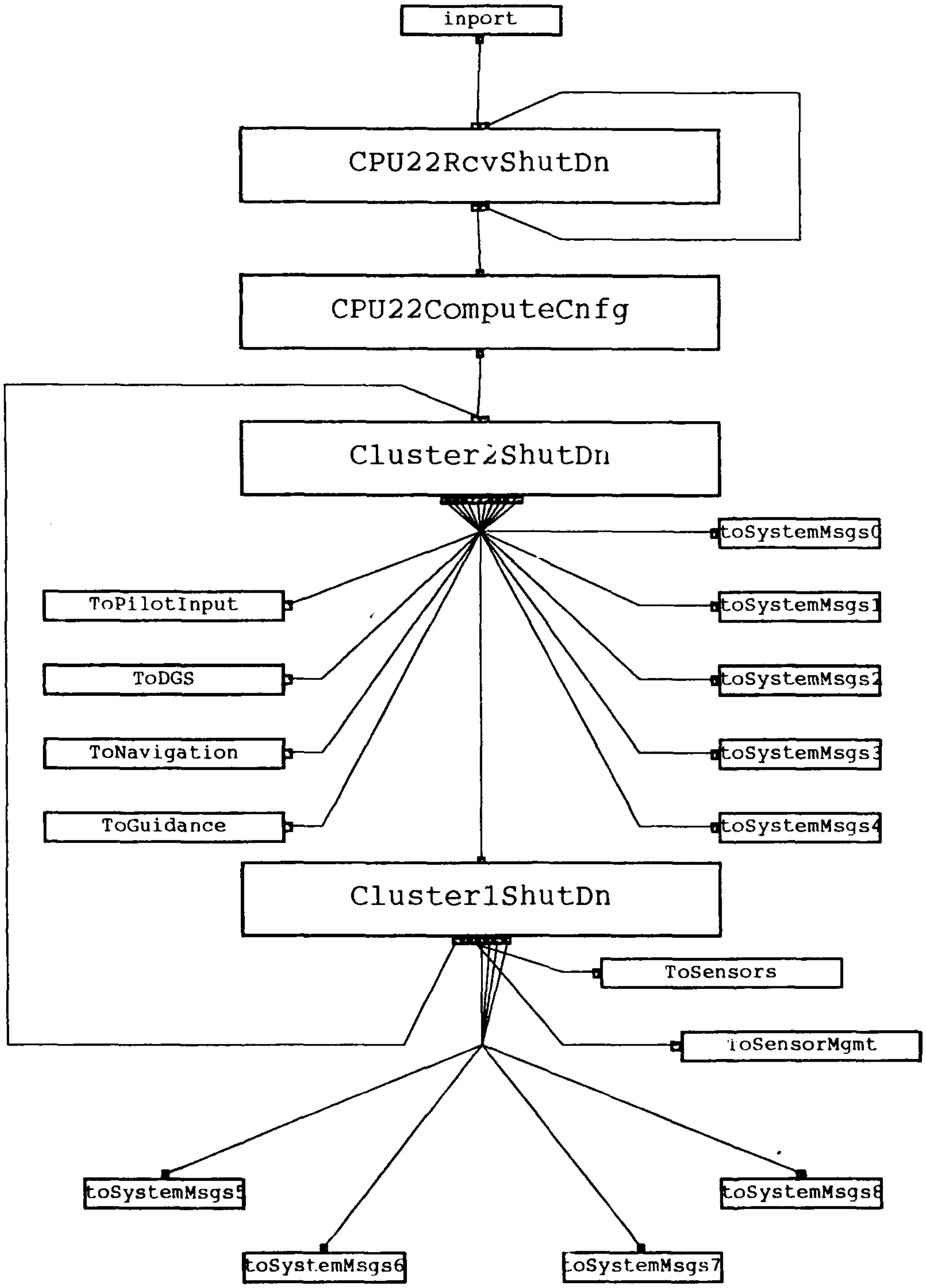

Figure A-52. To-Level Graph of Shutdown Process

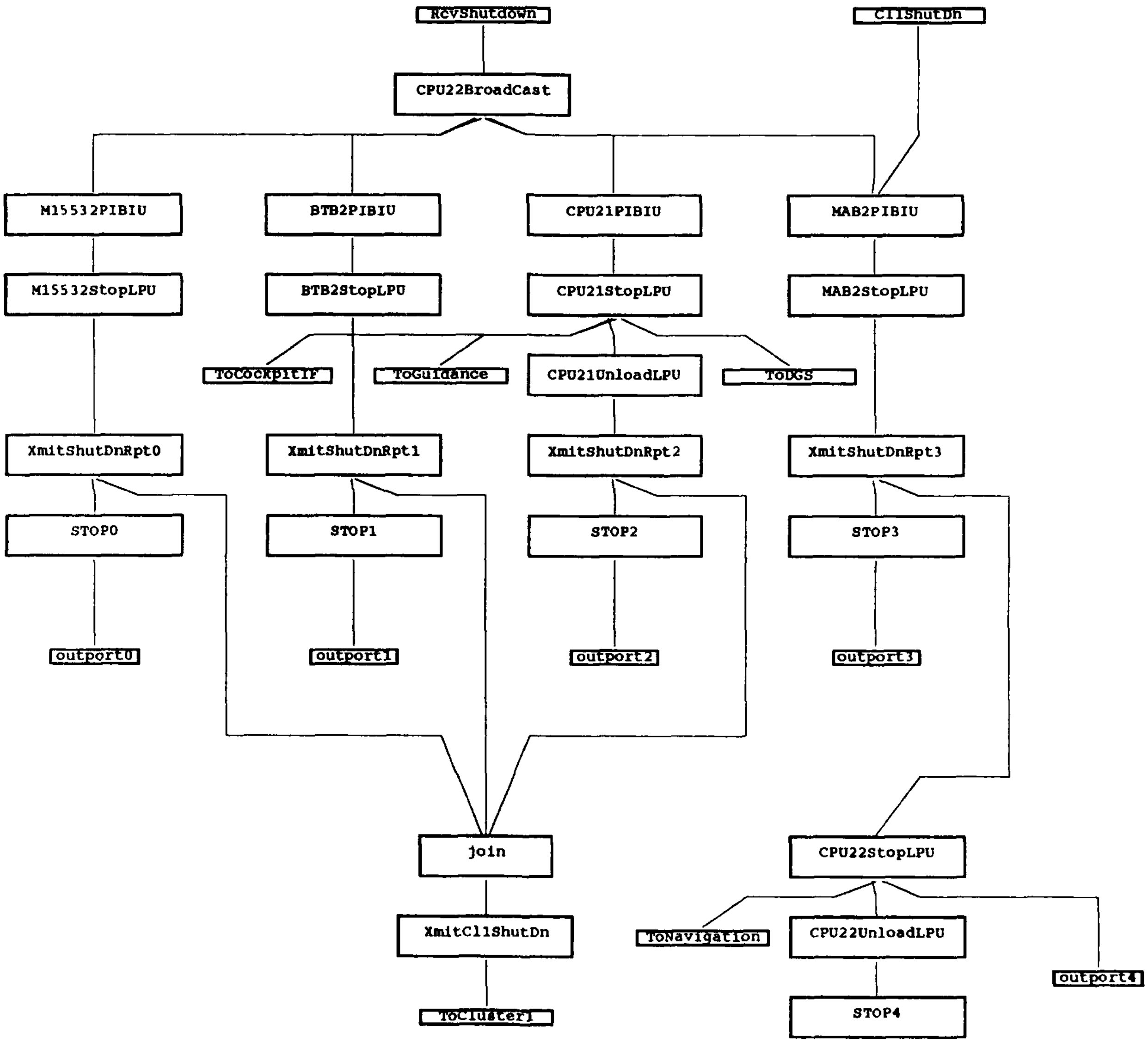

Figure A-53. Cluster 2 Shutdown

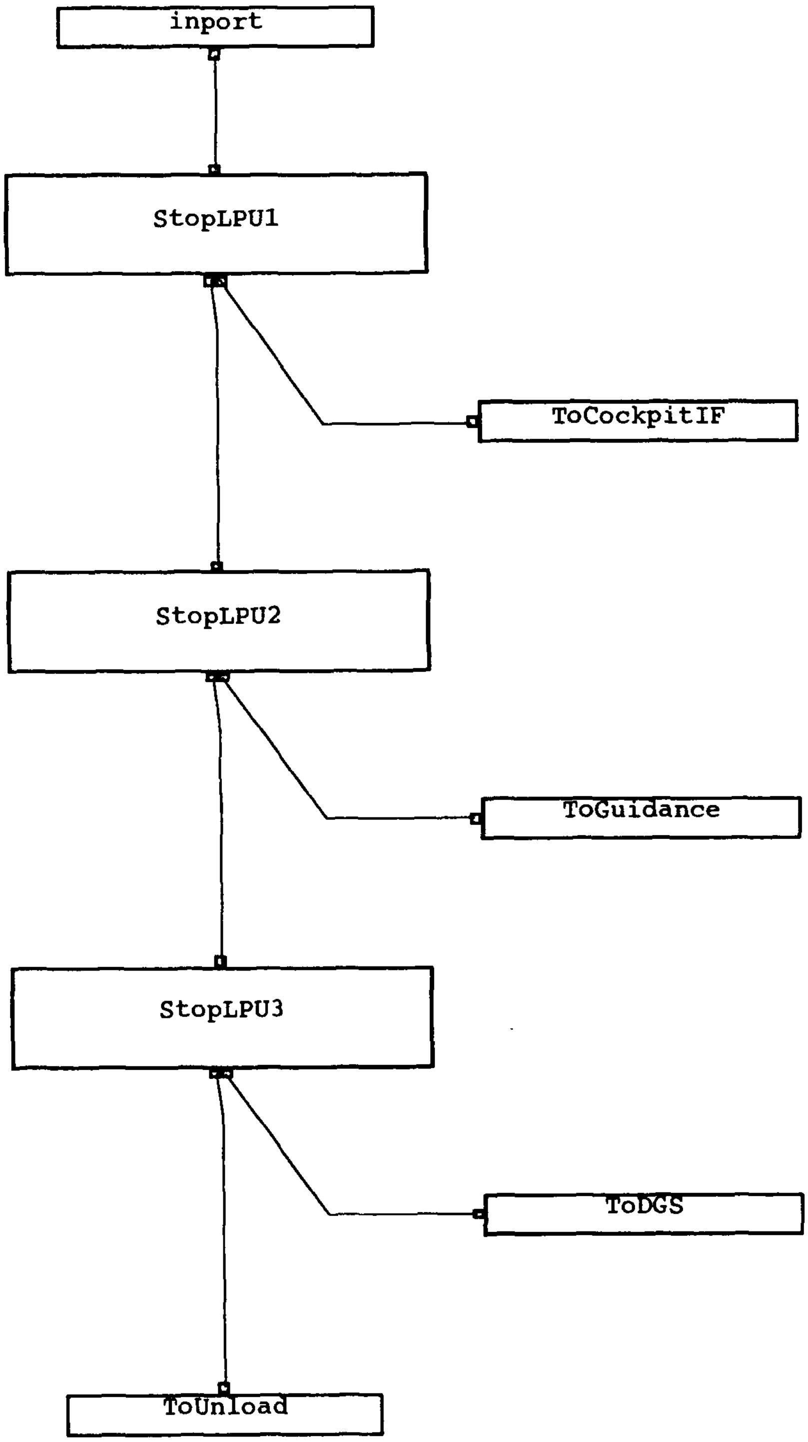

Figure A-54. CPU Stop LPUs

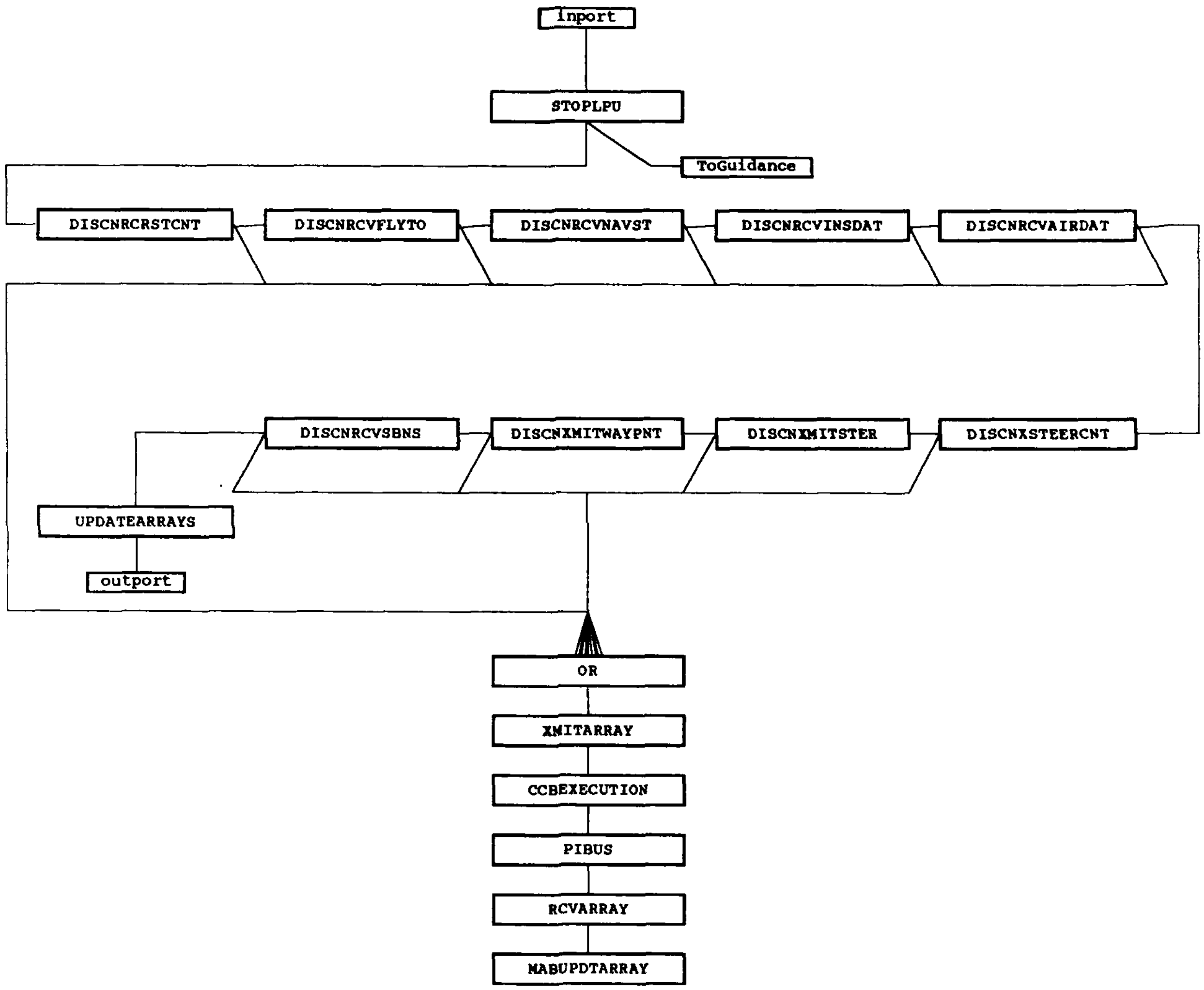

Figure A-55. Stop LPU

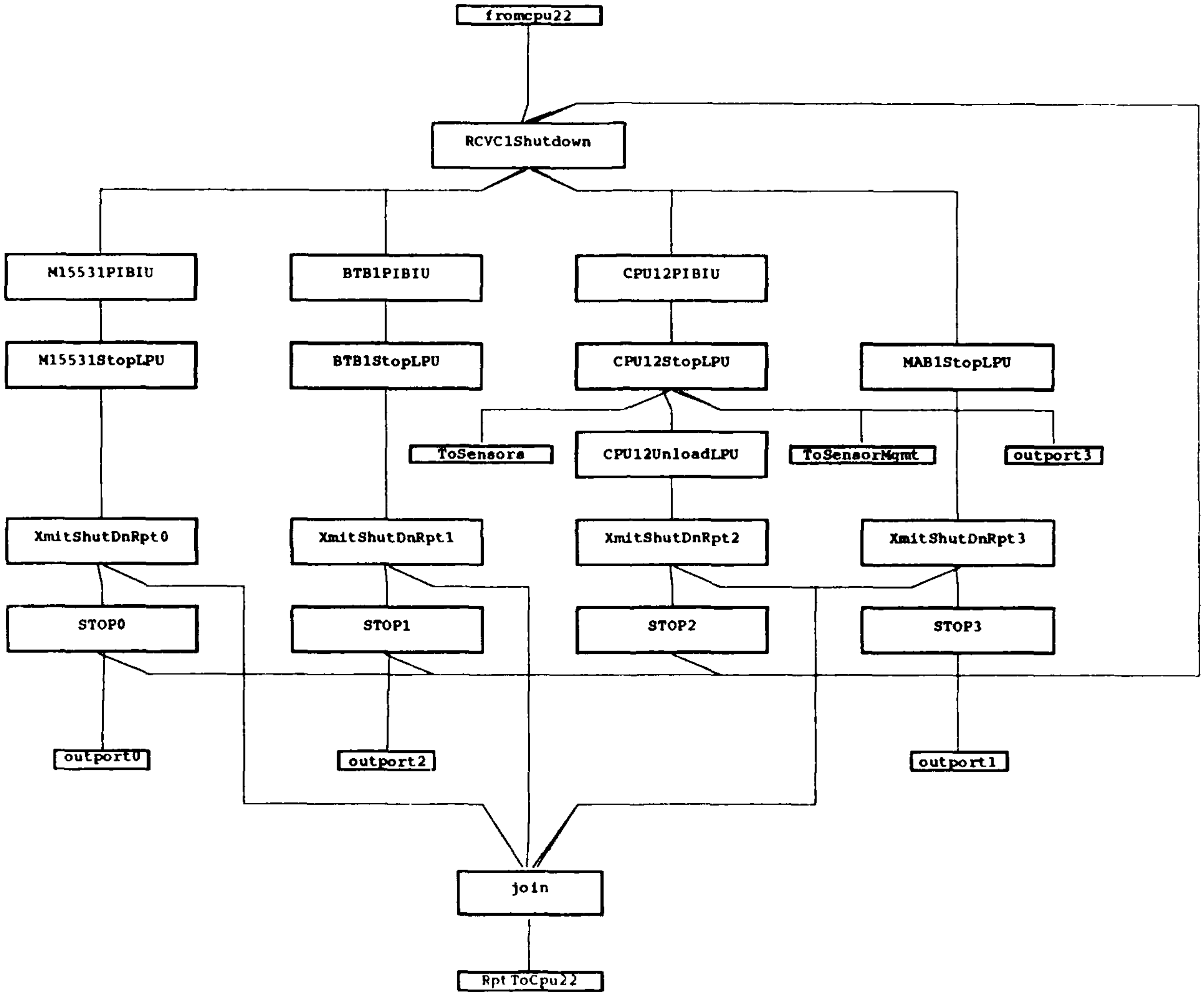

Figure A-56. Cluster 1 Shutdown

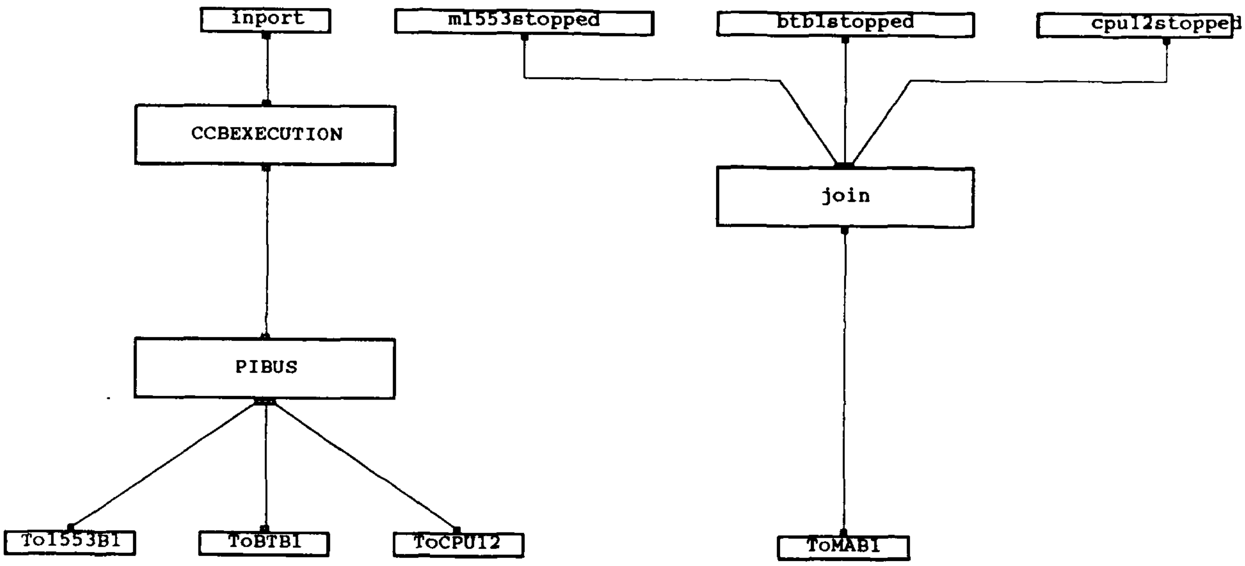

Figure A-57. Pass Cluster 1 Shutdown Message

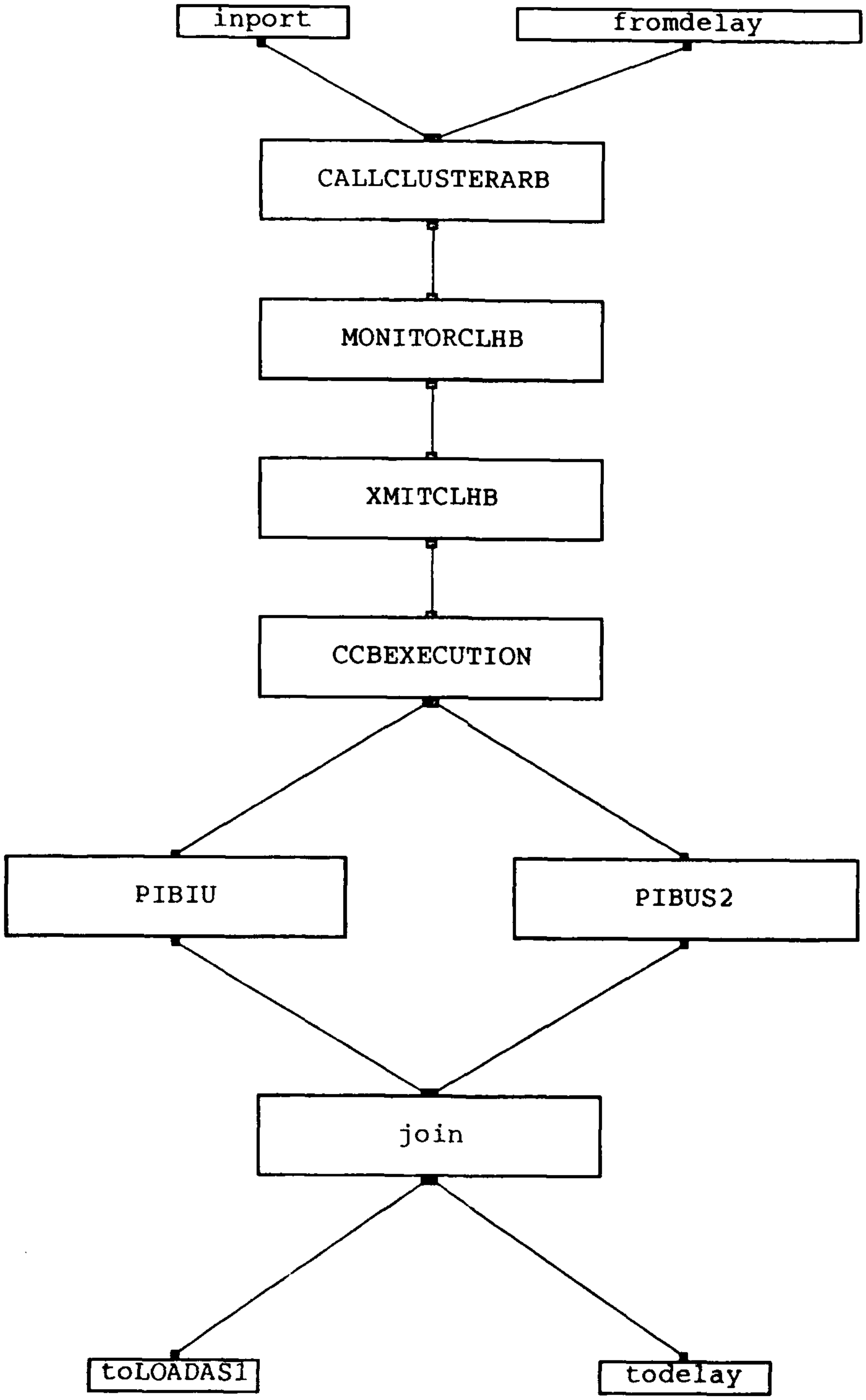

Figure A-58. Graph of a Cluster Arbitration Node

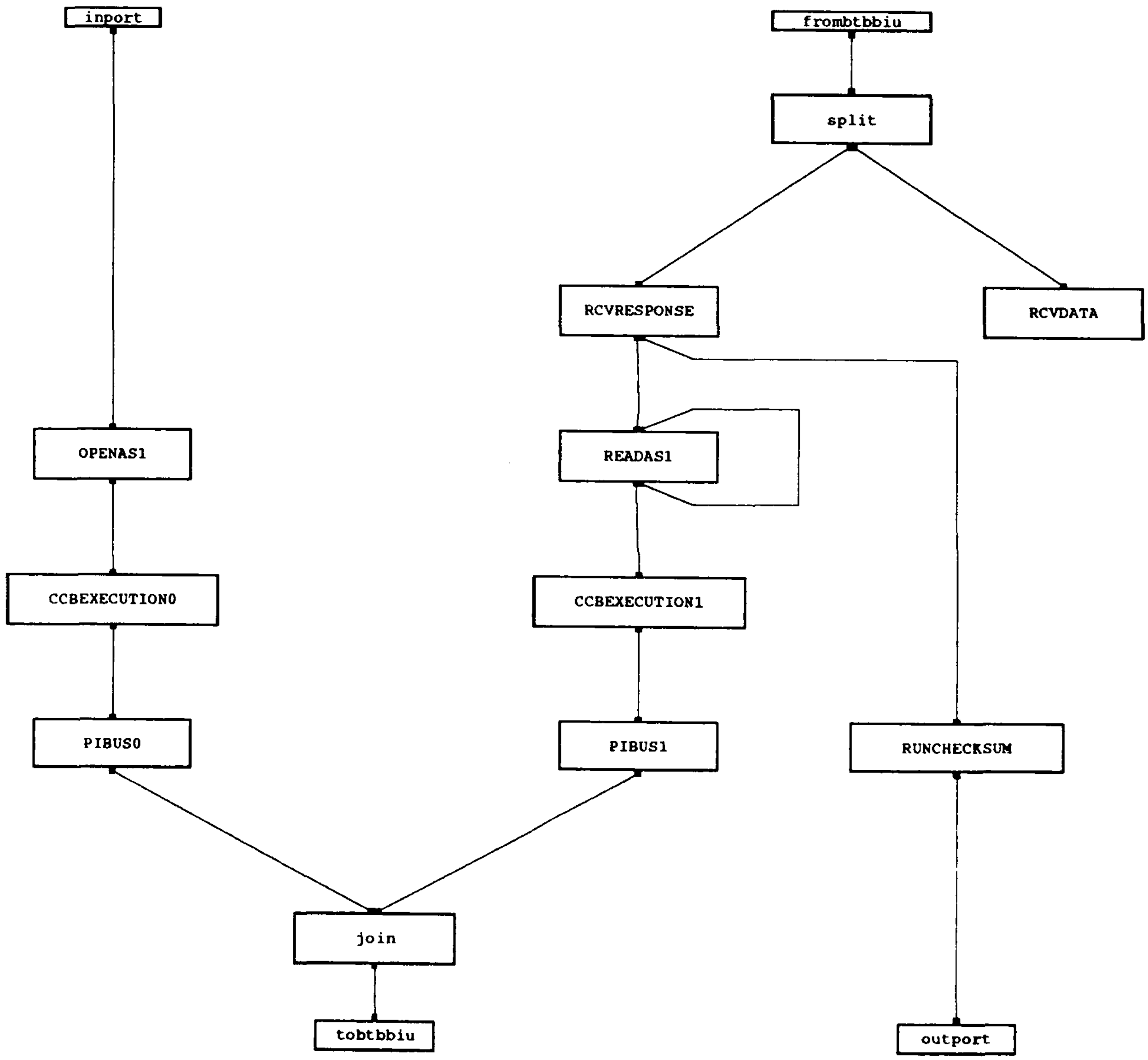

Figure A-59. Cluster Supervisor Load AS1

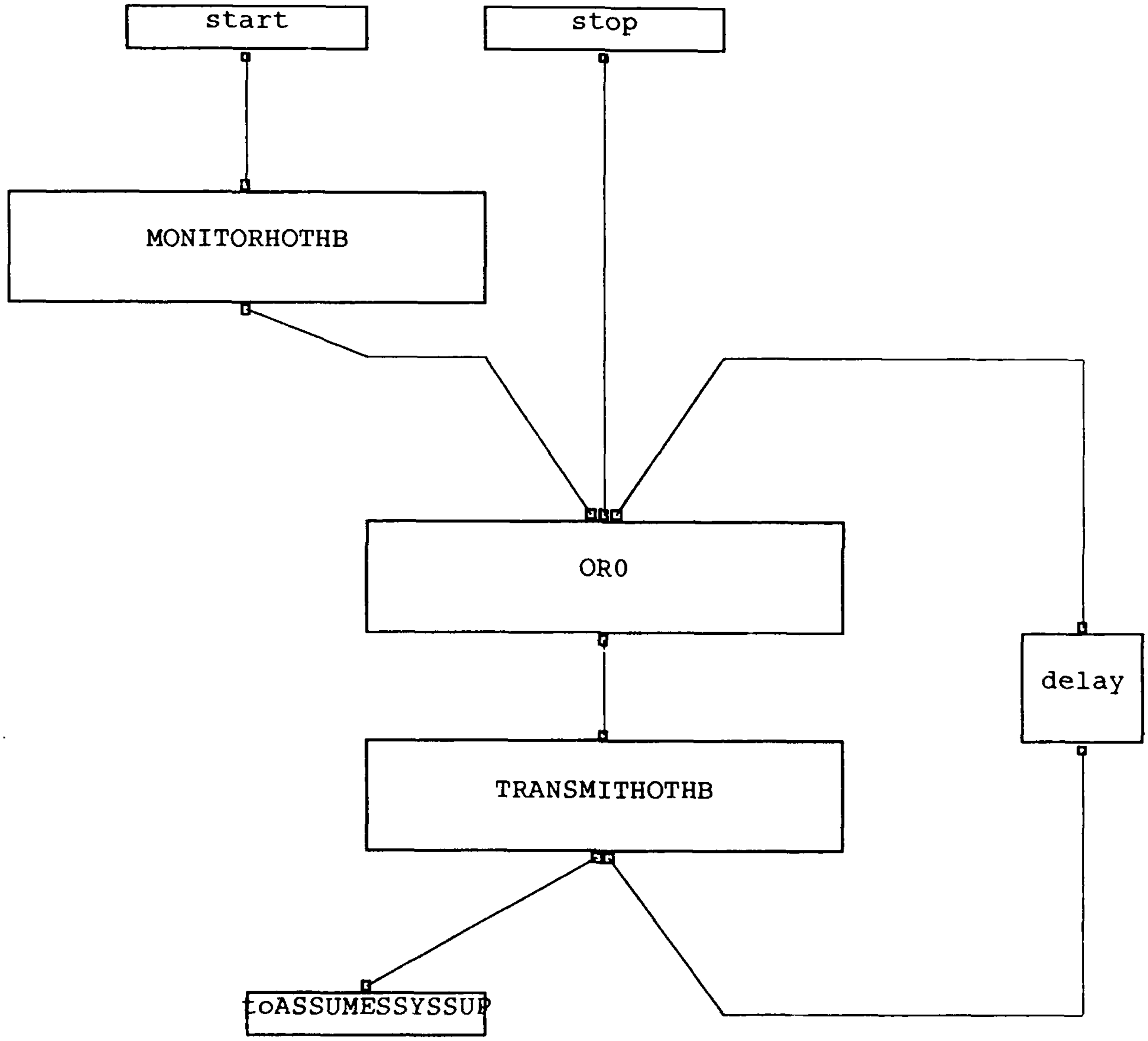

Figure A-60. Hot Backup Arbitration

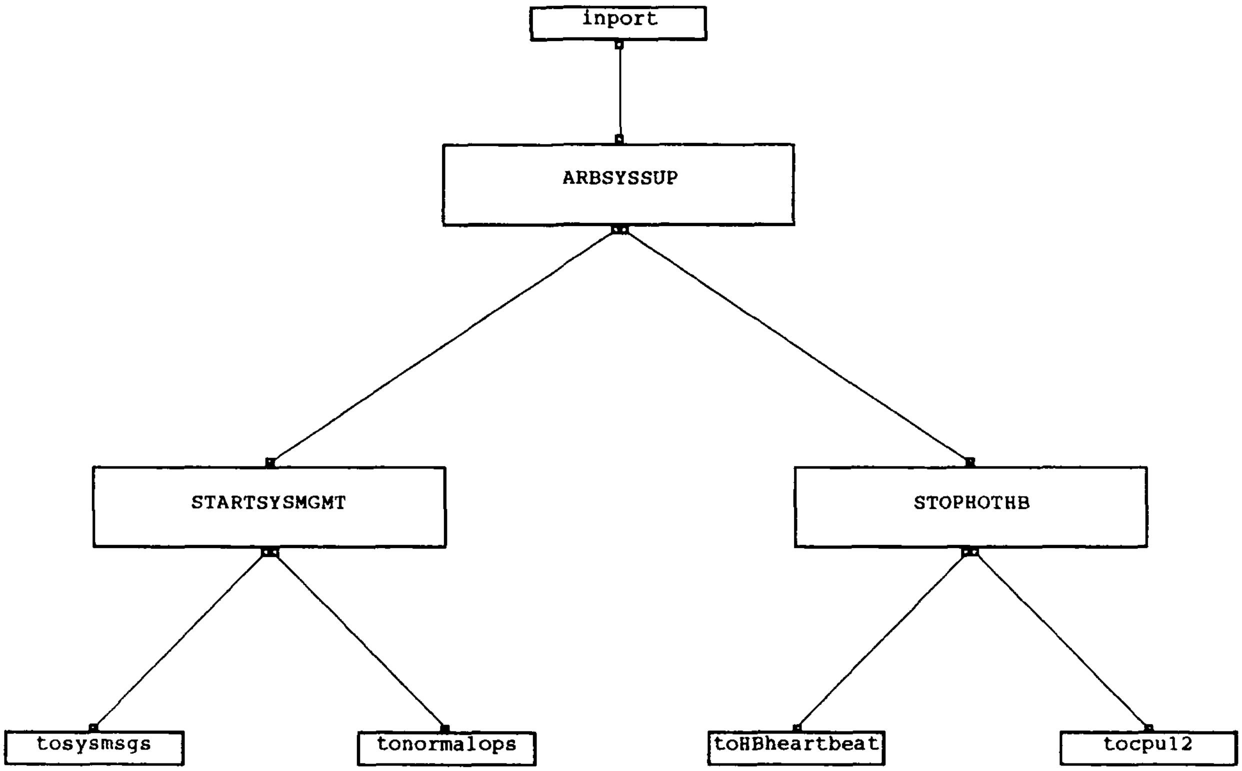

Figure A-61. Initiate the System Supervisor

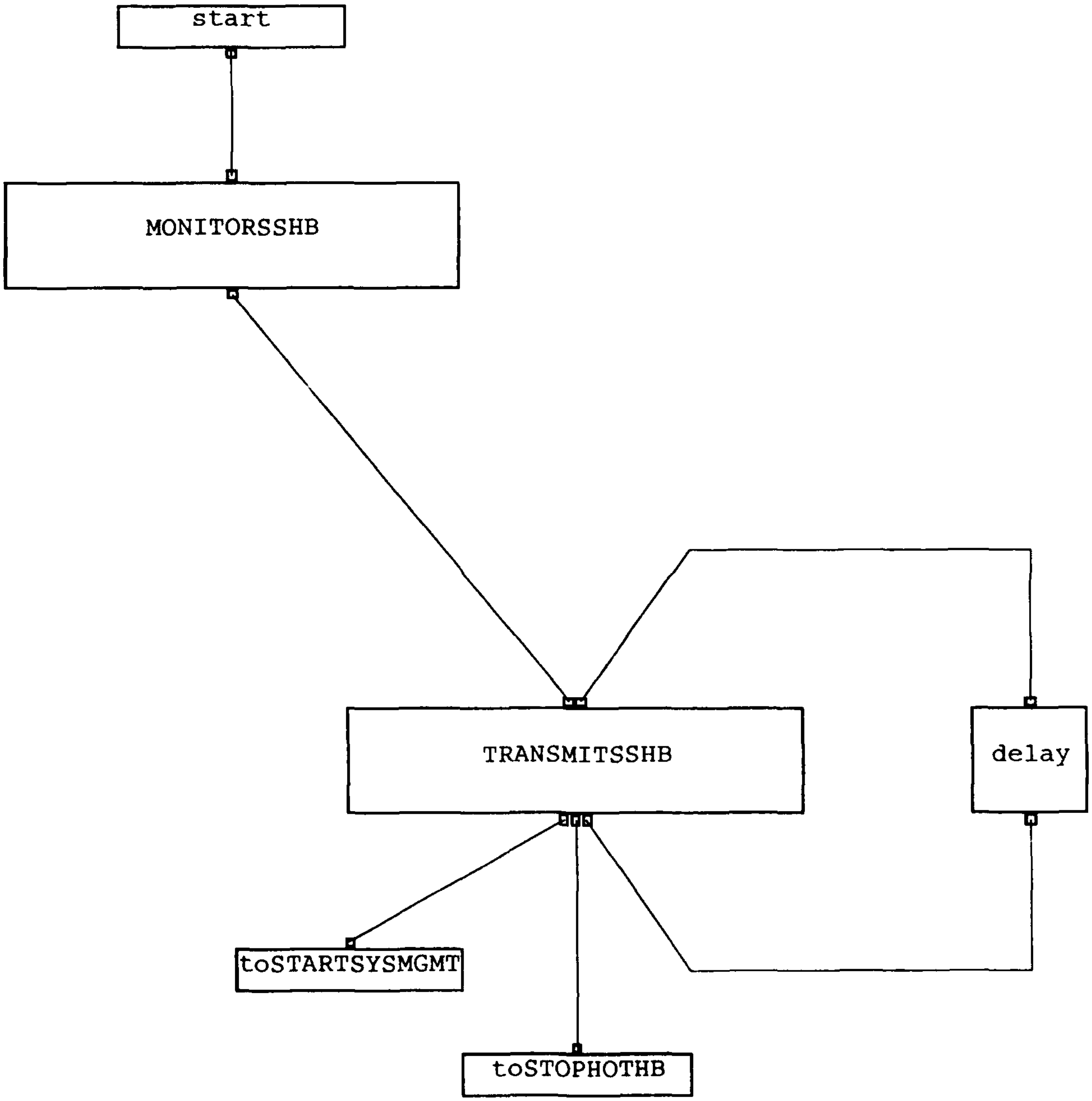

Figure A-62. System Supervisor Arbitration

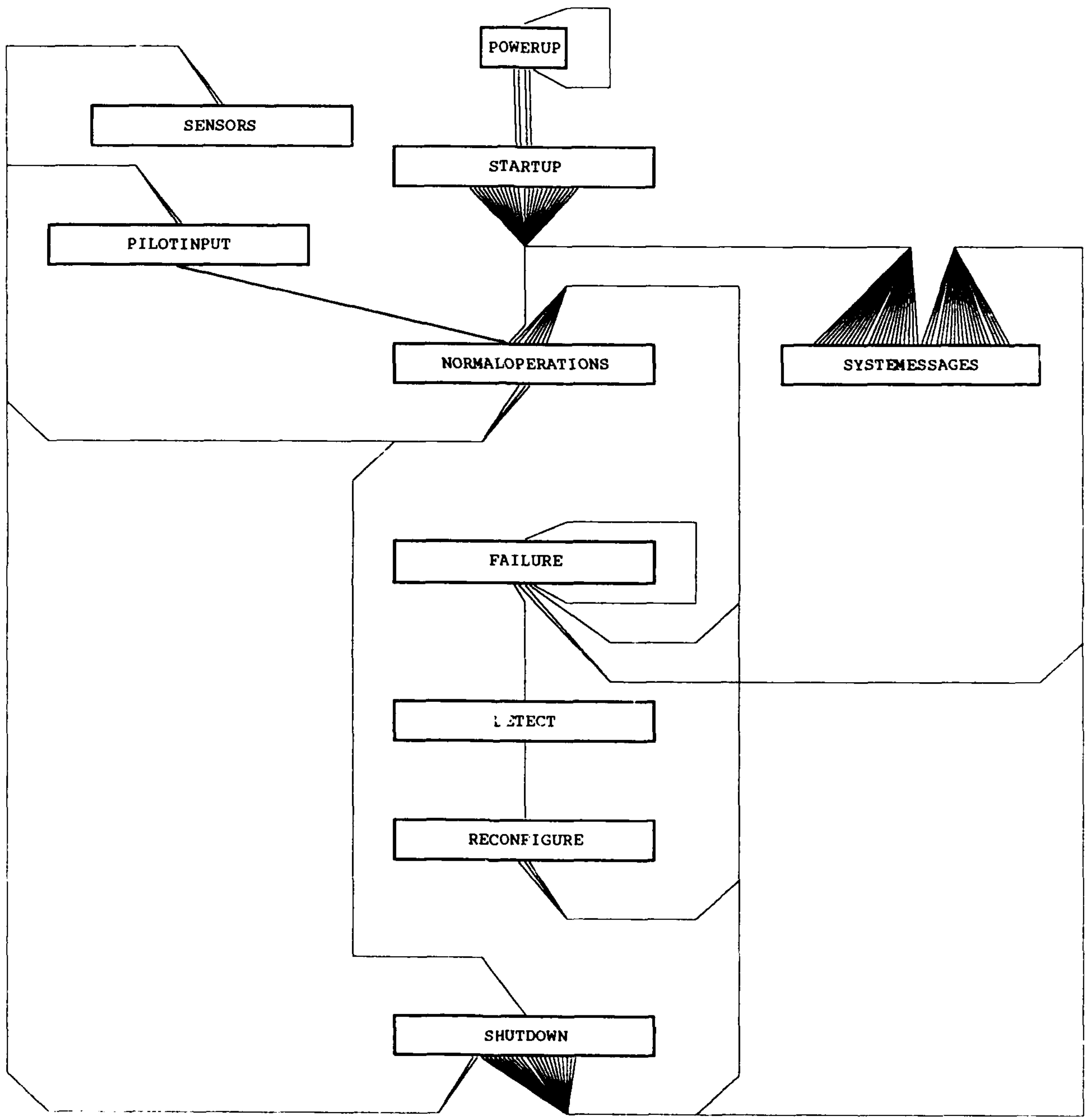

Figure A-63. Top-Level Graph for 4 Clusters

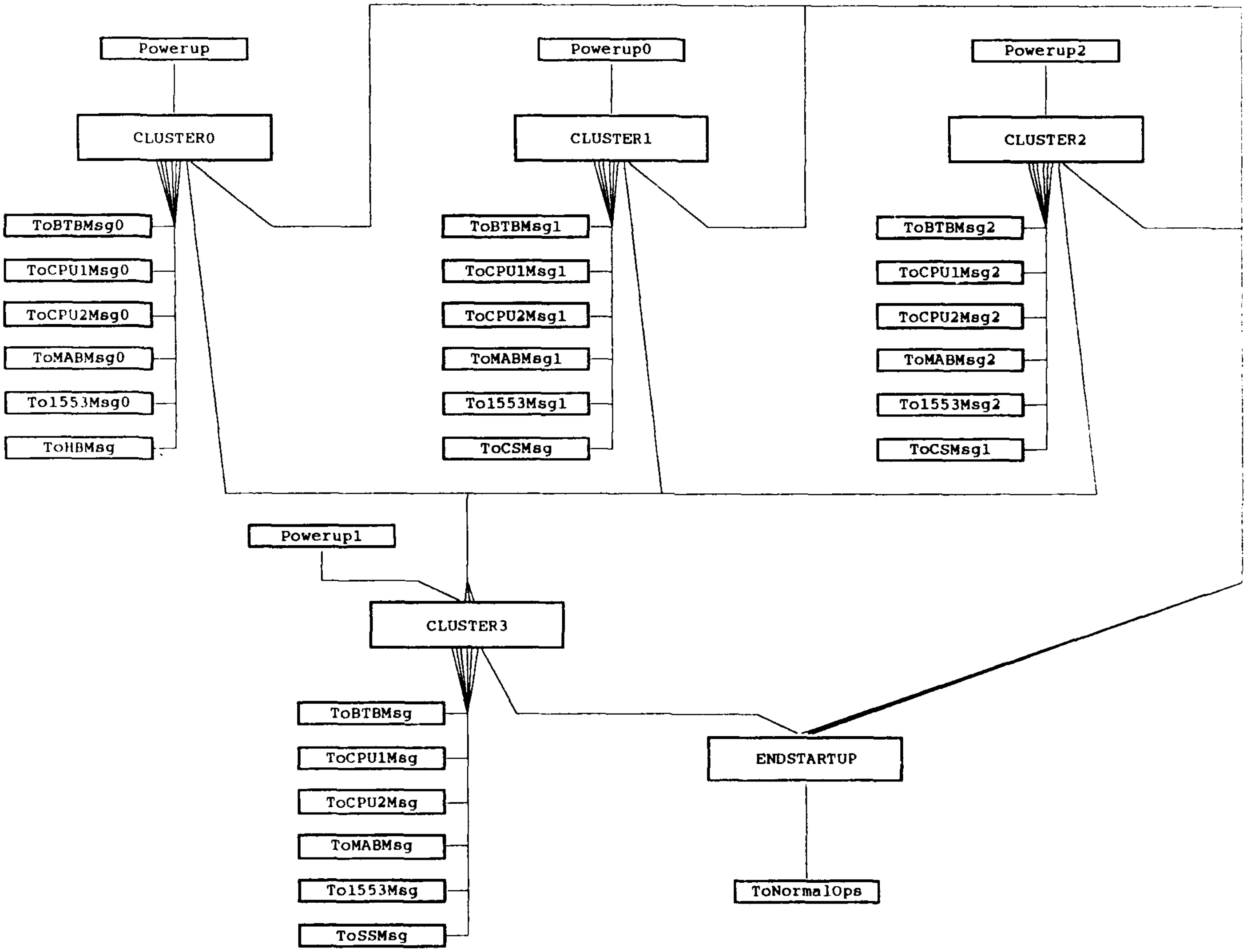

Figure A-64. Startup Graph for 4 Clusters

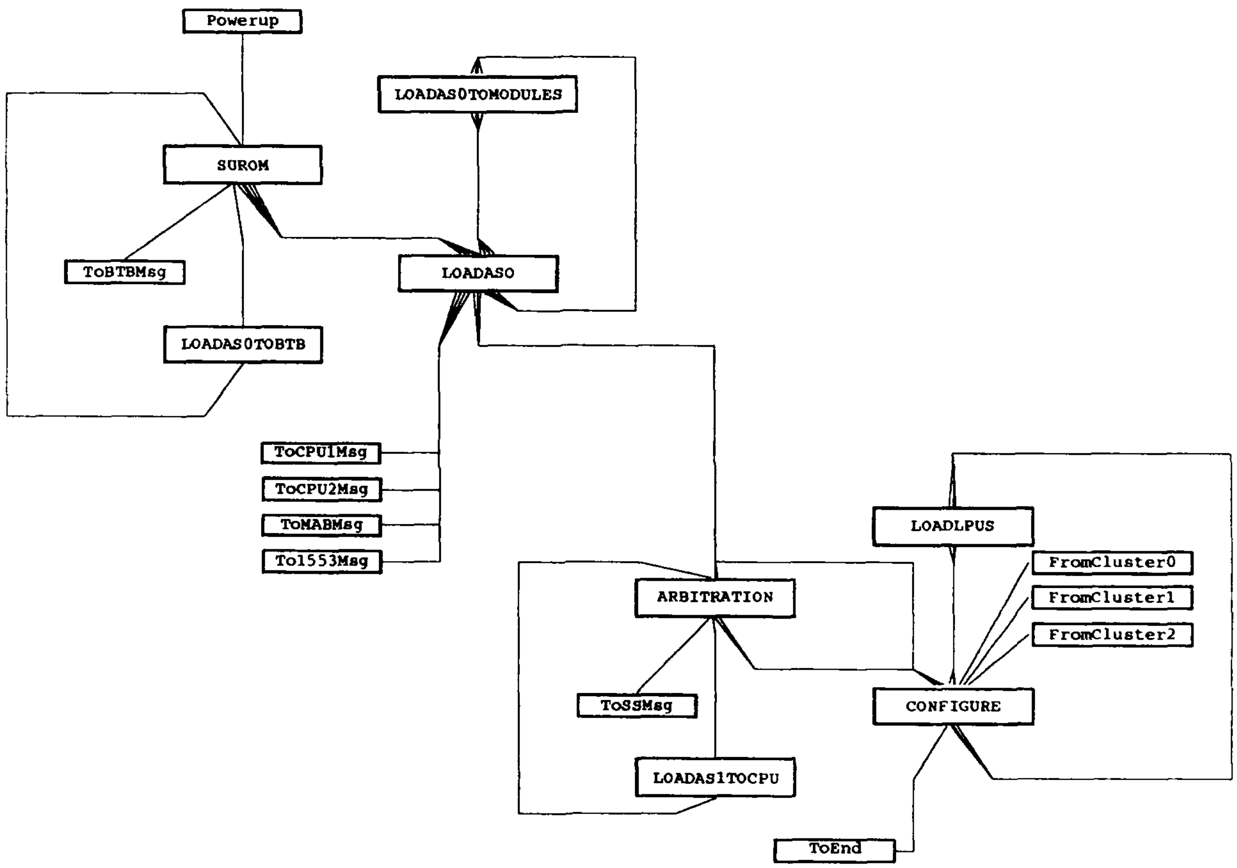

Figure A-65. Startup Graph of a Cluster

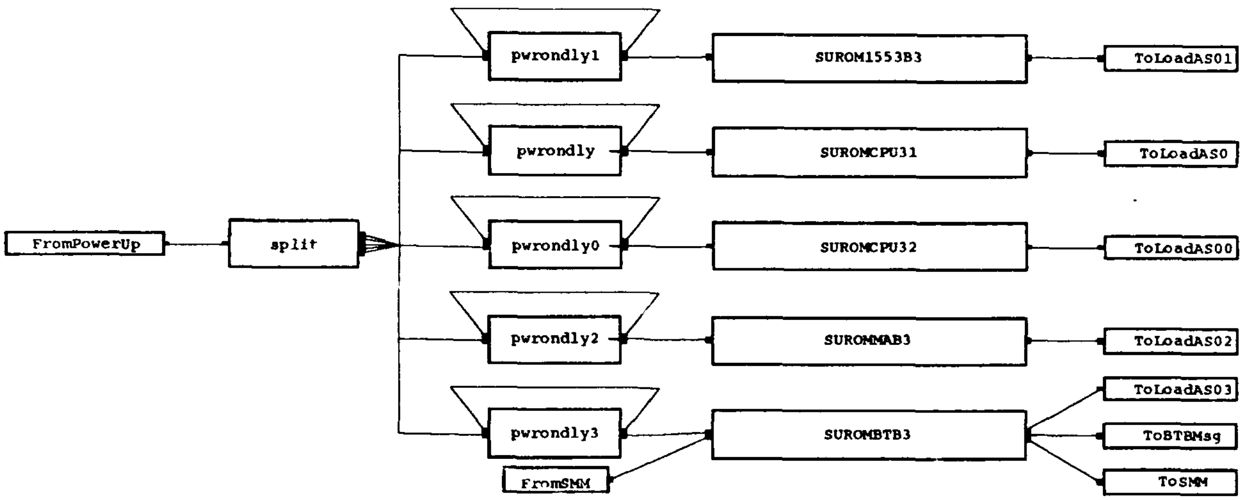

Figure A-66. Graph of **SUROM** and Active Load

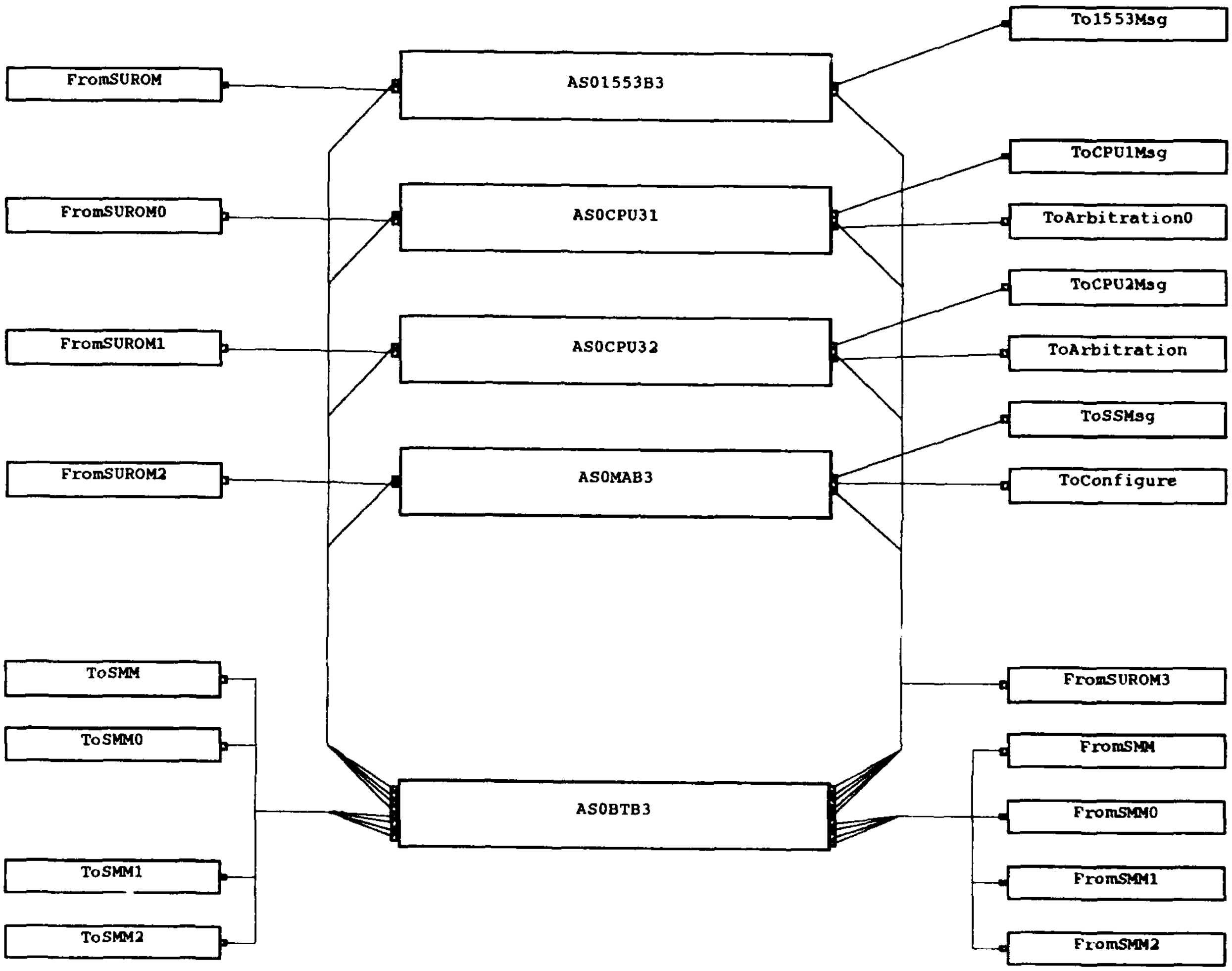

Figure A-67. Graph of AS0 Passive Load

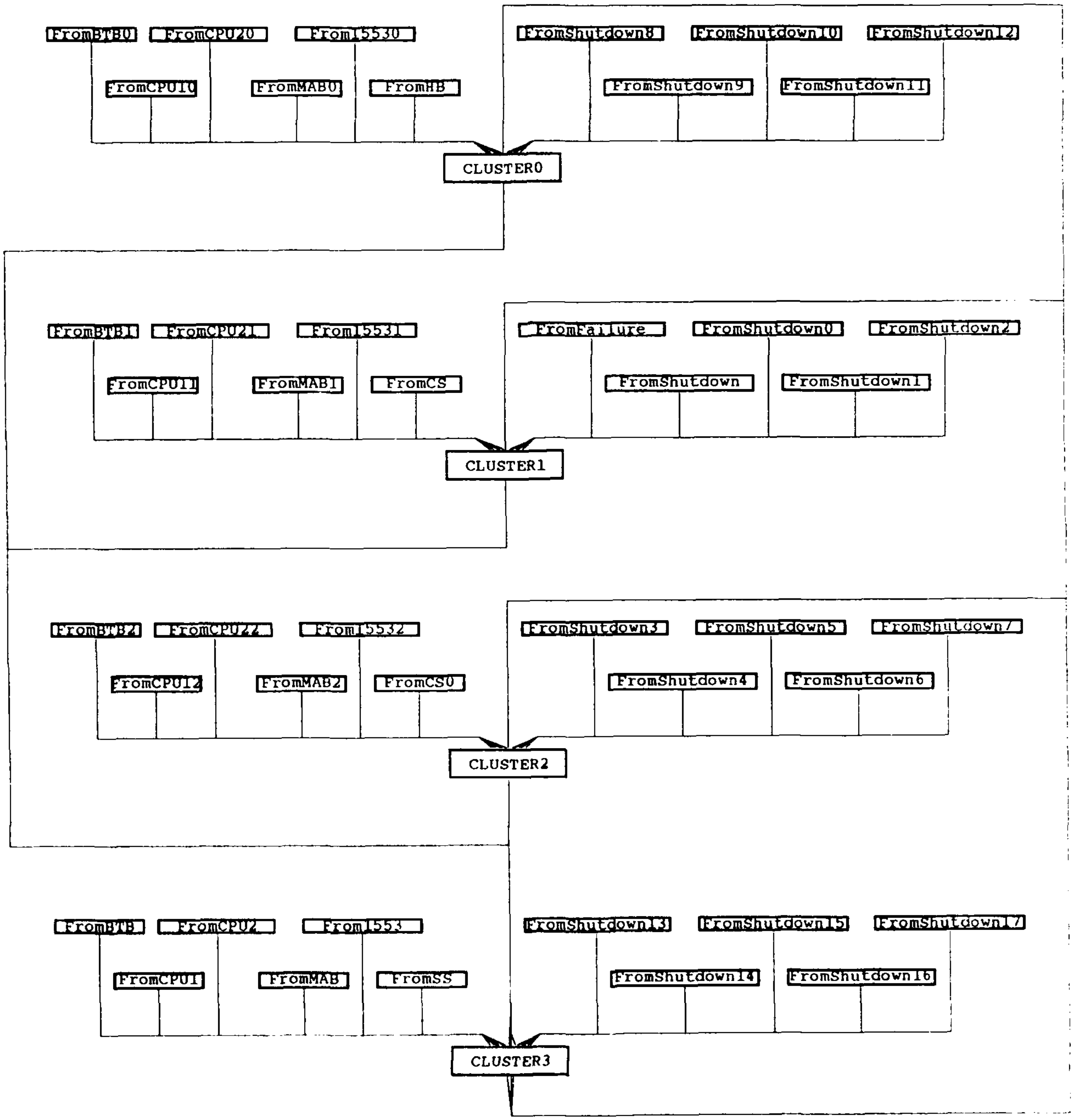

Figure A-68. System Messages with 4 Clusters

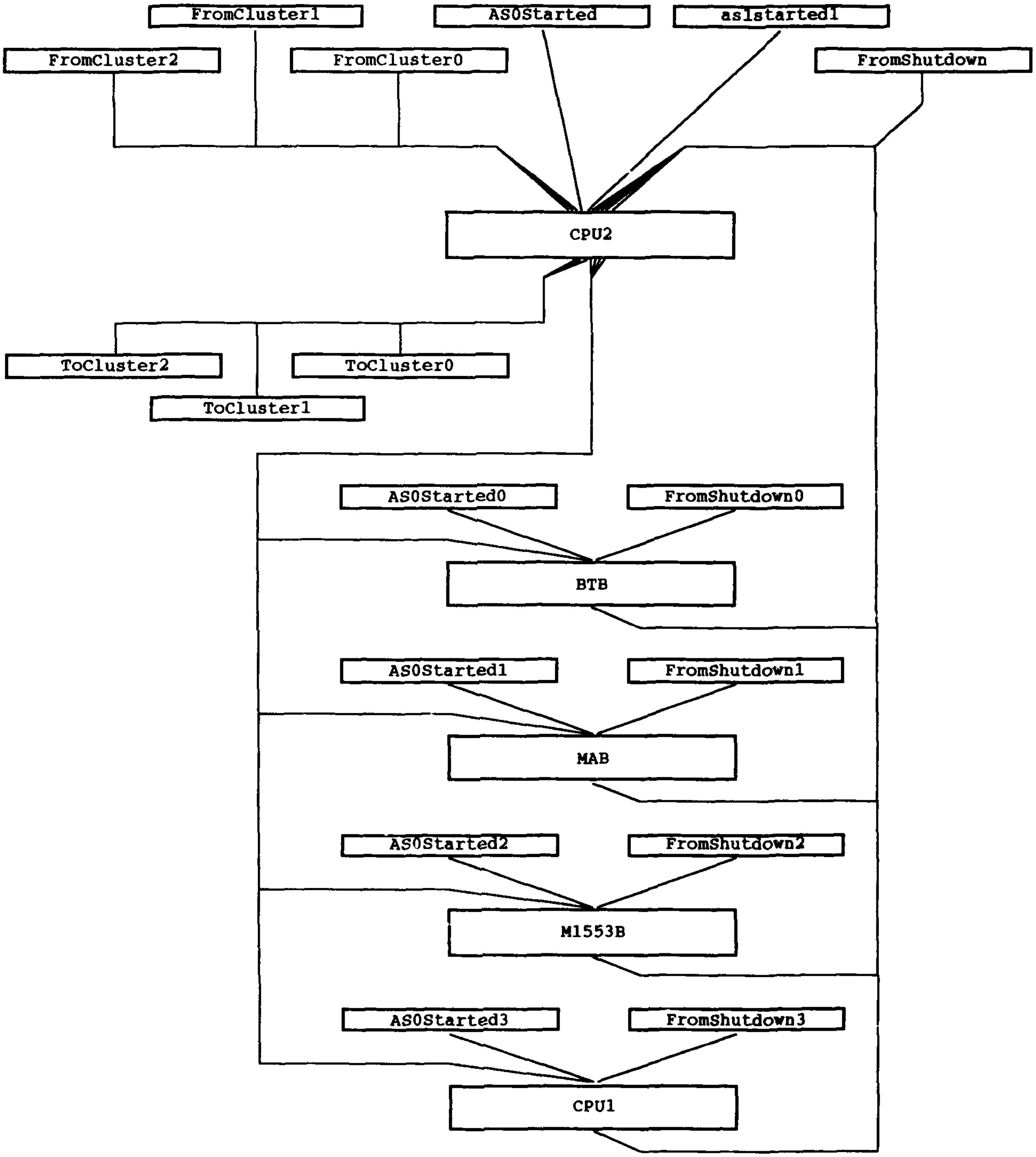

Figure A-69. System Messages: 1 Cluster

☆U.S. Government Printing Office:1992—648-127/62269